JAVACREAM

Training
Consulting
Projectmanagement

GIT

Vorstellungsrunde

JAVACREAM
Training
Consulting
Projectmanagement

- Name und Rolle im Unternehmen
- Themenbezogene Vorkenntnisse
- Aktuelle Problemstellung
- Individuelle Zielsetzung

Einführung

Erwartungshaltung an Versionsverwaltungssystem

- Verwaltung von “Meta-Informationen” wird übernommen
  - “wer hat wann was warum gemacht”
- Bestimmte Dateien werden in bestimmten Ständen zu einem Gesamt-Stand gruppiert
- Stände können parallel existieren
- Konsolidierung von Ständen
- Tooling, Historischer Verlauf, Unterschiede in Dateien/Ständen, …
- Gemeinsamer Zugriff durch einen Server (Authentifizierung, …)
- Team-Zusammenarbeit

Realisierung mit GIT

- Verwaltung von “Meta-Informationen” wird übernommen
  - “wer hat wann was warum gemacht”
- Bestimmte Dateien werden in bestimmten Ständen zu einem Gesamt-Stand gruppiert
- Stände können parallel existieren
- Konsolidierung von Ständen
- Tooling, Historischer Verlauf, Unterschiede in Dateien/Ständen, …
  - Konsole
- Gemeinsamer Zugriff durch einen Server (Authentifizierung, …)
- Team-Zusammenarbeit

Realisierung mit GIT, Git-Werkzeuge

- Verwaltung von “Meta-Informationen” wird übernommen
  - “wer hat wann was warum gemacht”
- Bestimmte Dateien werden in bestimmten Ständen zu einem Gesamt-Stand gruppiert
- Stände können parallel existieren
- Konsolidierung von Ständen
- Tooling, Historischer Verlauf, Unterschiede in Dateien/Ständen, …
  - Konsole, Integration ins Betriebssystem, Integration in Editoren und IDEs
- Gemeinsamer Zugriff durch einen Server (Authentifizierung, …)
- Team-Zusammenarbeit

Zu den Git-Werkzeugen

- Desktop
  - TortoiseGit
  - SourceTree
  - …
- PlugIns für Editoren
  - Eclipse
  - Visual Studio
  - Visual Studio Code
  - ...

Realisierung mit GIT, Git-Werkzeuge, Git Server

- Verwaltung von “Meta-Informationen” wird übernommen
  - “wer hat wann was warum gemacht”
- Bestimmte Dateien werden in bestimmten Ständen zu einem Gesamt-Stand gruppiert
- Stände können parallel existieren
- Konsolidierung von Ständen
- Tooling, Historischer Verlauf, Unterschiede in Dateien/Ständen, …
  - Konsole, Integration ins Betriebssystem, Integration in Editoren und IDEs, Web Console
- Gemeinsamer Zugriff durch einen Server (Authentifizierung, …)
- Team-Zusammenarbeit

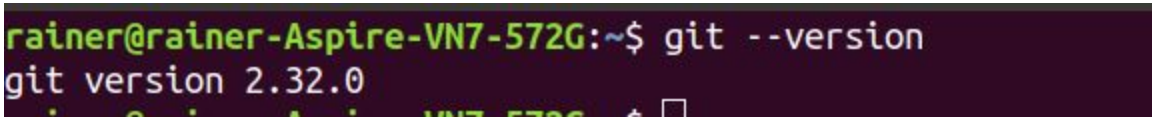
Git Server

- GitHub
  - Server laufen in der Microsoft-Cloud
  - Öffentliche Ablage ist kostenlos, private Bereiche Lizenz-pflichtig
- BitBucket
  - Atlassian
  - Cloud-Service + Betrieb auf eigenen Servern
- GitLab
  - gitlab.com
  - Cloud-Service + Betrieb auf eigenen Servern
- Azure DevOps
  - Microsoft-Cloud

First Contact

Git Bash

- Terminal-Fenster mit Git-Unterstützung
- Das Kommando “git” steht hierin zur Verfügung
  - Das ist **kein simples Command Line Interface**, das mit einem Git-Server kommuniziert
    - Es gibt keinen laufenden Git-Server-Prozess, kein Dämon, …
  - git --version
  -



Der erste Git-Befehl: config

- git config <scope> <key-hierarchie> <value>
  - <scope>
    - local
    - global (User-Profile)
    - system
  - <key-hierarchie>
    - “.” trennt die Hierarchie-Ebenen
  - <value>
    - irgendwas, Leerzeichen etc. aber in Anführungszeichen setzen
- git config --global user.name “Rainer Sawitzki”
- git config --global user.email training@rainer-sawitzki.de
- Auslesen git config --get user.name

Erstes Arbeiten mit Git

Einrichten eines Git-Projektverzeichnisses


```
mkdir first
cd first
```

```
git init
```

```
git status
        Fehlerfrei
```

Normales Verzeichnis -> Git-Projektverzeichnis

Git Workspace

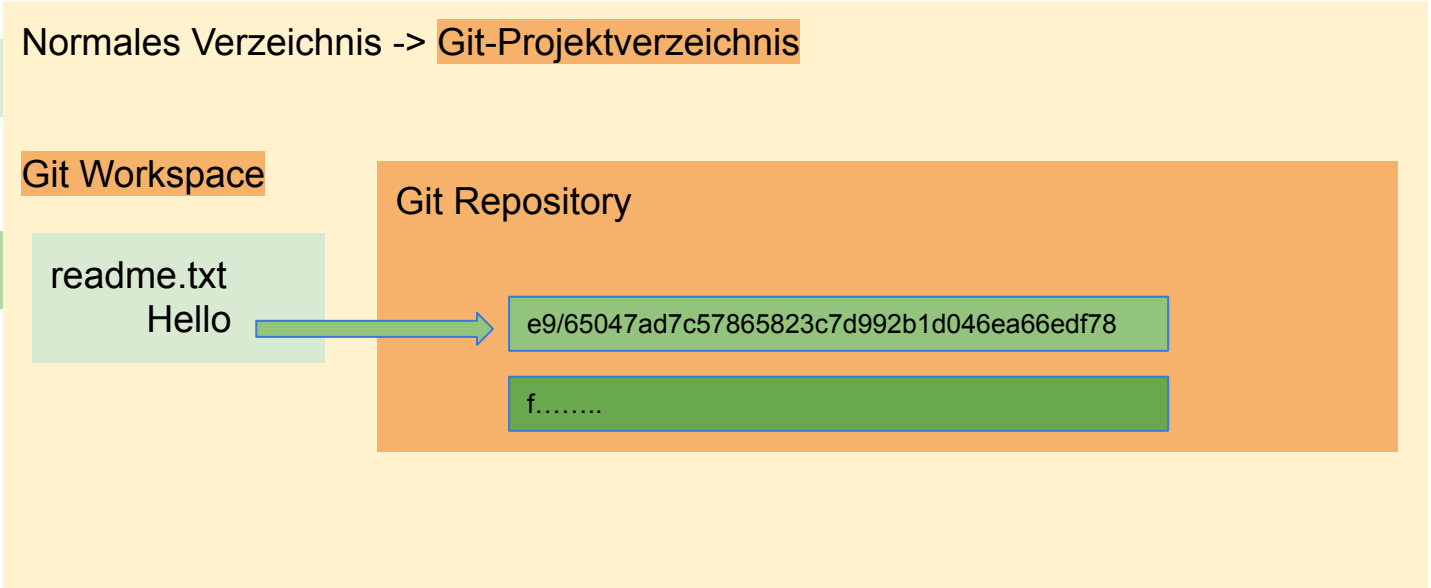
Git Repository

Hinweis:
In der Praxis entspricht diese Sequenz einem
git clone server-repo first

Hinzufügen von Informationen


```
echo Hello > readme.txt
```

```
git add readme.txt
```



Exkurs

- Was ist diese Datei “e9/65047ad7c57865823c7d992b1d046ea66edf78”?
  - Binärformat, das heißt Informationen sind in dieser Datei abgelegt worden
  - Der Dateiname ist
    - ~~UID, weltweit eindeutiger Zufallswert~~ oder
    - Ein berechneter Hash-Wert, berechnet aus dem Inhalt “Hello”

Interpretation

- e9/65047ad7c57865823c7d992b1d046ea66edf78
- Wenn in irgendeinem Git-Repository dieser Hash-Wert existiert ist es mit höchster Wahrscheinlichkeit eine Datei mit dem Inhalt “Hello”
  - Wahrscheinlichkeit einer Kollision ist absurd gering

Commits definieren Stände


```
echo Hello > readme.txt
```

```
git add readme.txt
```

```
git commit -m “warum?”
```

Normales Verzeichnis -> Git-Projektverzeichnis

Git Workspace

readme.txt
Hello

Git Repository

Index / Staging Area

e9/65047ad7c57865823c7d992b1d046ea66edf78

f……..

Commit

Objekt

Aufbau des Commit-Objekts

- Liste der Dateien / Informationen, die zu diesem Stand=Commit gehören
- Committer
  - user.name + user.email
- Timestamp
- Commit-Message
  - Angabe über Option git commit -m “...”
  - Ohne Option öffnet sich ein Editor-Fenster
    - vim (Drücke “i” für den Insert-Modus, Fertig: ESC, Eingabe von :wq
    - nano
    - git config --global core.editor notepad

Interpretation

- Ist in einem Repository ein Commit mit dem Hash ee23e95a40724fb1ad5b119d2ed9e8f7c069813e vorhanden:
  - Rainer Sawitzki hat am um eine Commit erstellt mit der Message add content und den Dateien readme.txt(Hello) content.txt(Hugo)

Objekte in Git

- Allgemein
  - Dateien, die über einen Hashwert identifiziert sind
- Typen
  - Content- oder BLOB-Objekte
    - Diese repräsentieren Inhalte
  - Tree-Objekte
    - Pfad-Informationen
      - Diese werden erst beim commit erzeugt
  - Commit-Objekte
    - Diese repräsentieren einen Stand

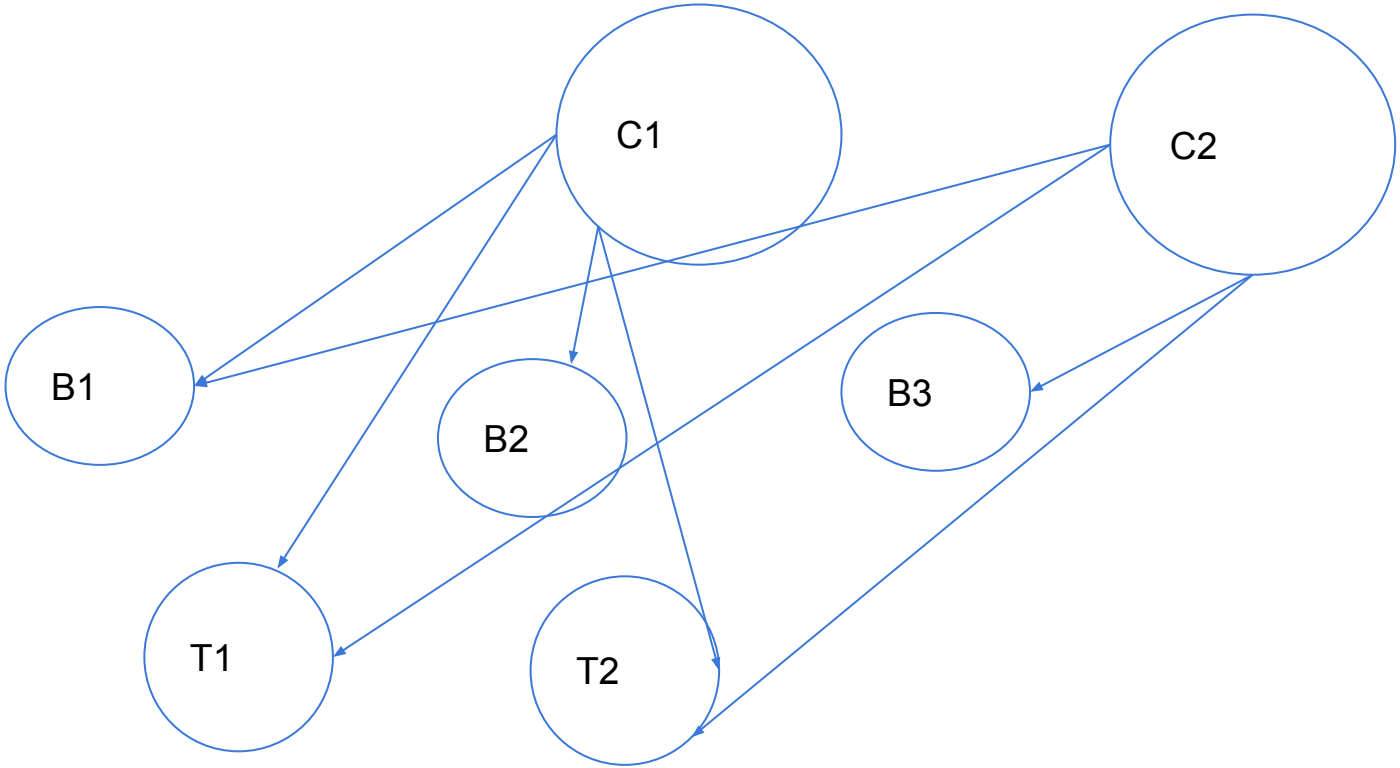
Arbeiten mit git

- Sie arbeiten normal im Workspace
  - Änderungen, neue Dateien, löschen
- Zweistufiger Prozess
  - Welche Dateien sollen hinzugenommen werden?
    - git add <file> | <directory>
      - inklusive Jokerzeichen
      - git add .
  - Erzeugen des Commit-Objekts
    - git commit -m "..."
- Vorsicht
  - "es gibt doch git commit -a"
  - -a = --all
    - --all bezieht sich nur auf Dateien, die schon in der Staging-Area waren

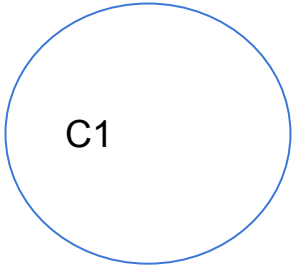
Exkurs: .gitignore

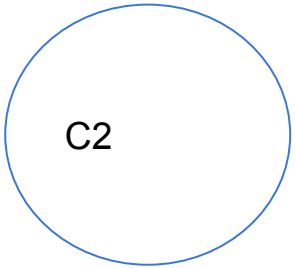
- Textdatei als Bestandteil des Workspaces
  - oder eines Unterverzeichnisses
- In dieser Textdatei werden Regeln hinterlegt, die Dateien ausschließen
  - Unterverzeichnis-.gitignores werden gemerged mit denen der Ober-Verzeichnisse
- Hinweis
  - Standard-Namen (*.bak, …) werden automatisch ausgeschlossen

Die Git-Objekte im Detail



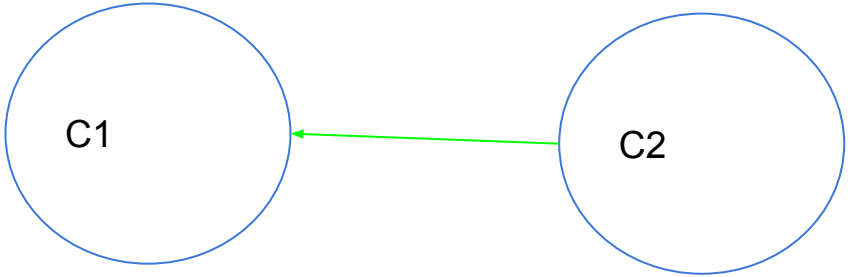
Weitere Visualisierung des Repos





Es werden nur noch die Commit-Objekte gezeichnet

Weitere Visualisierung des Repos:
Korrekt



Aufbau des Commit-Objekts

- Liste der Dateien / Informationen, die zu diesem Stand=Commit gehören
- Committer
  - user.name + user.email
- Timestamp
- Commit-Message
- Hash des Vorgänger-Commits

Interpretation

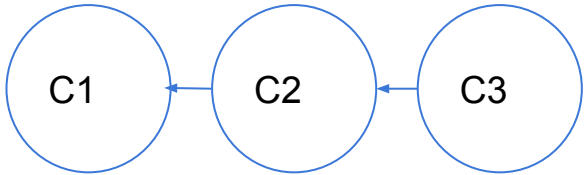
- Ist in einem Repository ein Commit mit dem Hash 9e7f4ac1208a5ceaa5f36e4d4a96276b9318c5fa vorhanden:
  - Rainer Sawitzki hat am um eine Commit erstellt mit der Message ... und den Dateien … ausgehend vom Commit mit dem Hashwert …. der dann wiederum den Hash ee23e95a40724fb1ad5b119d2ed9e8f7c069813e mit Rainer Sawitzki hat am um eine Commit erstellt mit der Message add content und den Dateien readme.txt(Hello) content.txt(Hugo)
-

Git-Technologie = Blockchain

- Durch dieses Arbeiten mit über Hash-Werte verketteten Informationen entsteht eine unmodifizierbare, nicht nachträglich änderbare Historie von Informationen
- Grundlage dieser Technologie sind die so genannten Merkle-Trees

Aktueller Stand

Fokus auf Hash-Werte, “Nerd-Modus”



Pragmatisches Arbeiten: Alias-Namen auf Commit-Objekte

- 2 Szenarien für die Einführung eines Namens
  - Definition eines fixen Standes
    - Typischerweise Versionen
      - v1.0, v1.1-Milestone1
    - “HeuteMorgen”
  - Bennenung einer gerade laufenden Aktion innerhalb eines sich entwickelnden Projektes
    - Typischerweise ist das der Name eine Aktion, einer Ticket-Nummer
      - “implement_feature1”, Jira-Ticket 0815
    - “working”, “experiment”, ...

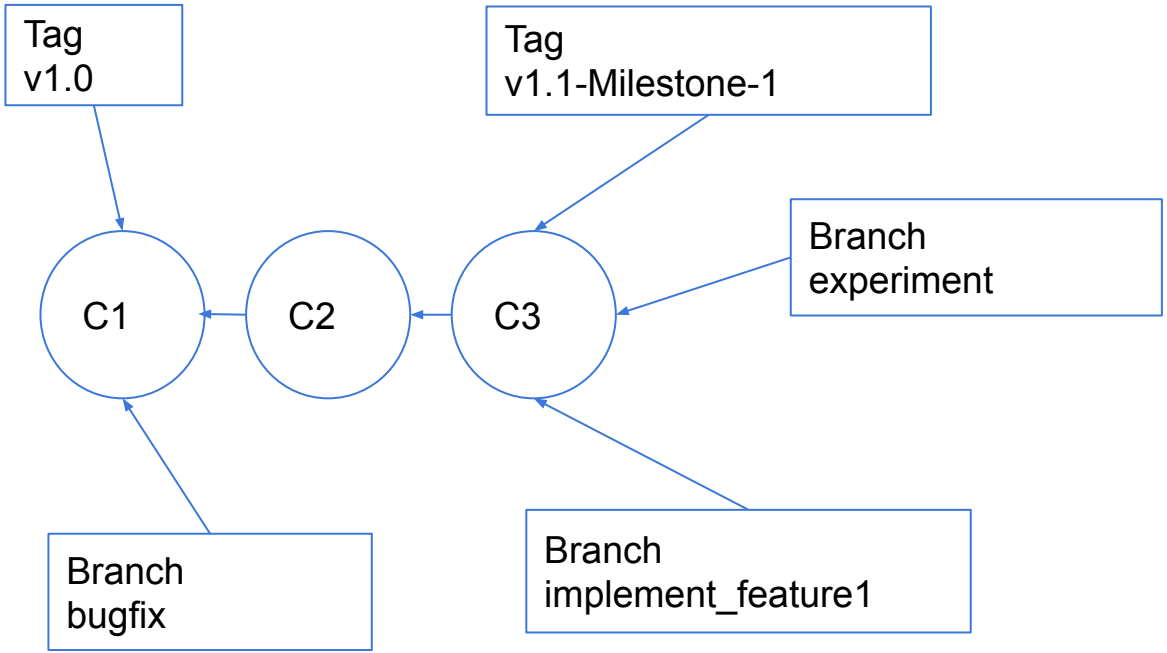
Pragmatisches Arbeiten: Alias-Namen auf Commit-Objekte mit Git

- 2 Szenarien für die Einführung eines Namens
  - Definition eines fixen Standes
    - Typischerweise Versionen
      - v1.0, v1.1-Milestone1
    - “HeuteMorgen”
    - Git: Tag
  - Bennenung einer gerade laufenden Aktion innerhalb eines sich entwickelnden Projektes
    - Typischerweise ist das der Name eine Aktion, einer Ticket-Nummer
      - “implement_feature1”, Jira-Ticket 0815
    - “working”, “experiment”, …
    - Git: Branch

Umsetzung in Git

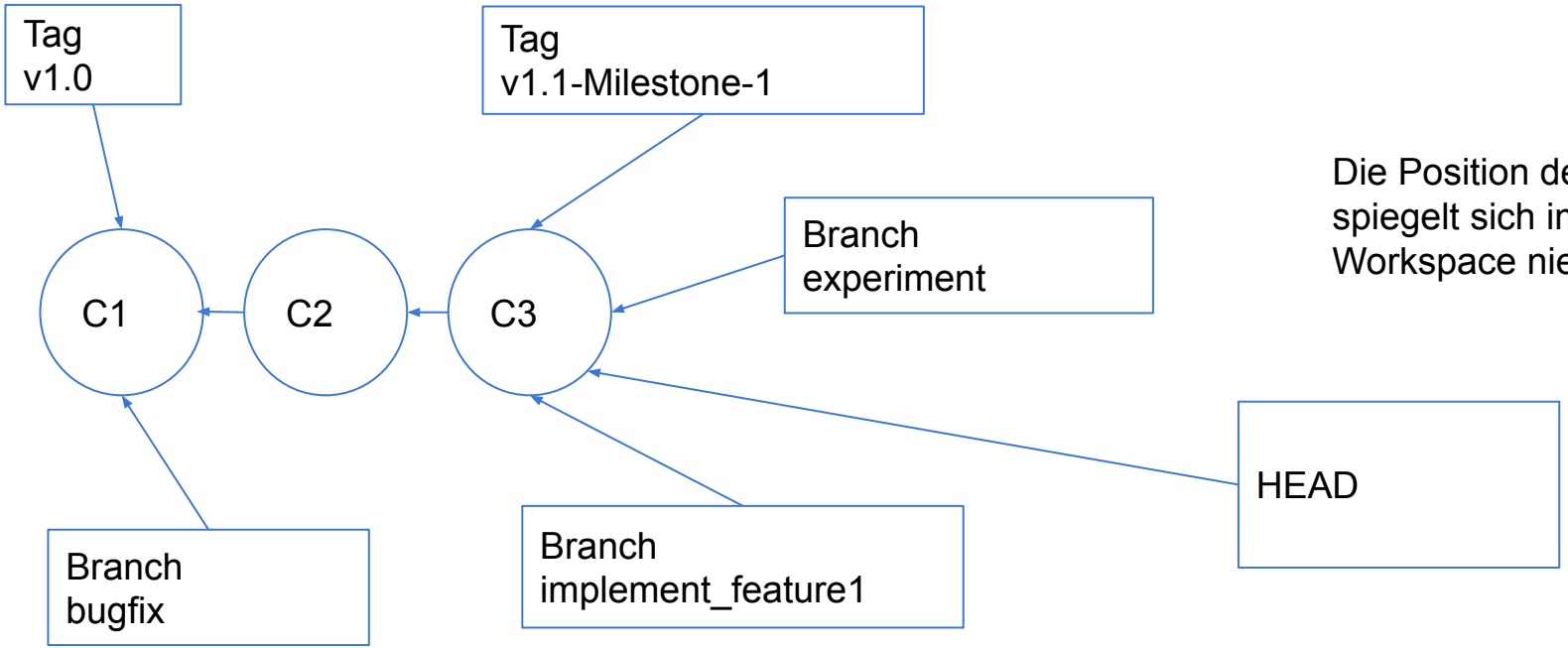
- Tags und Branches sind **trivial** in Erzeugung und Verwaltung
  - git tag | branch new_name
    - Erzeugung
  - git tag | branch -d name
    - Löschen
  - git tag | branch --list
    - Liste
  - git branch -m old_name new_name

Tags und Commits: Beispiel



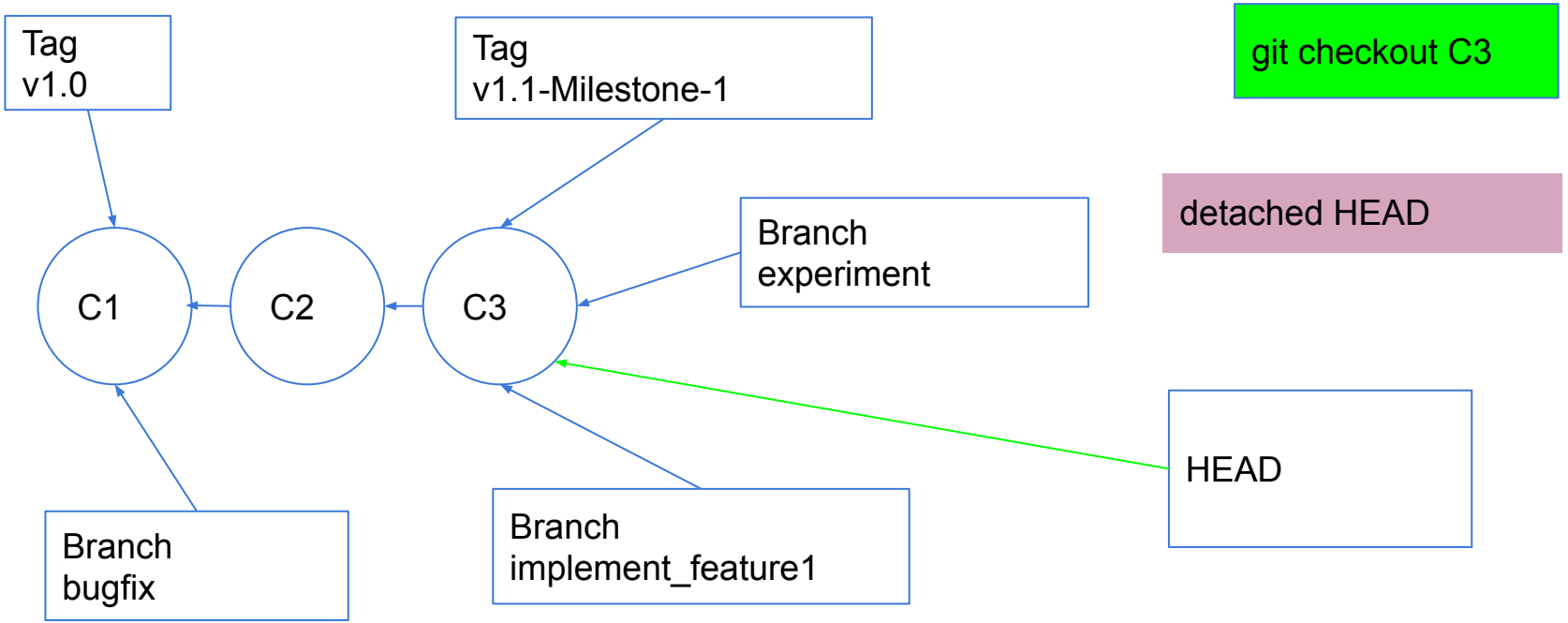
Tags und Branches sollen einen Überblick über den Stand des Projektes liefern

Aktuelle Position: HEAD

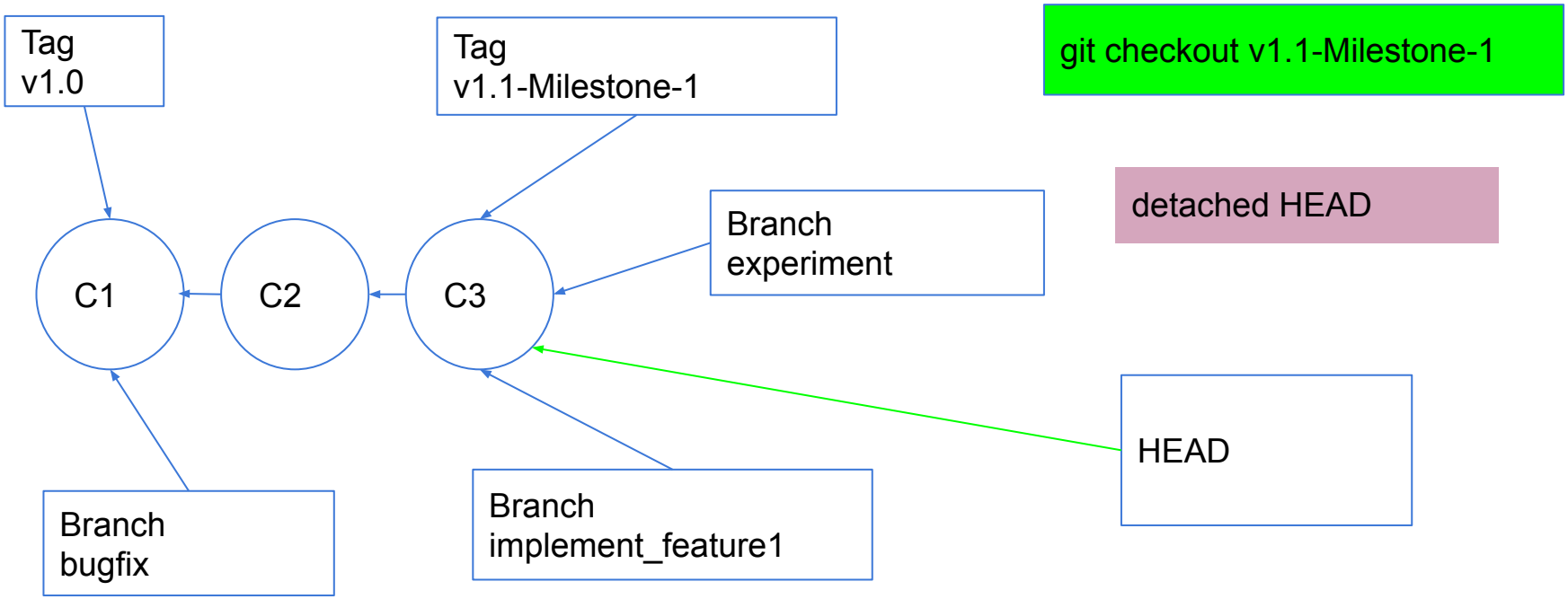


Die Position des HEAD spiegelt sich im Workspace nieder

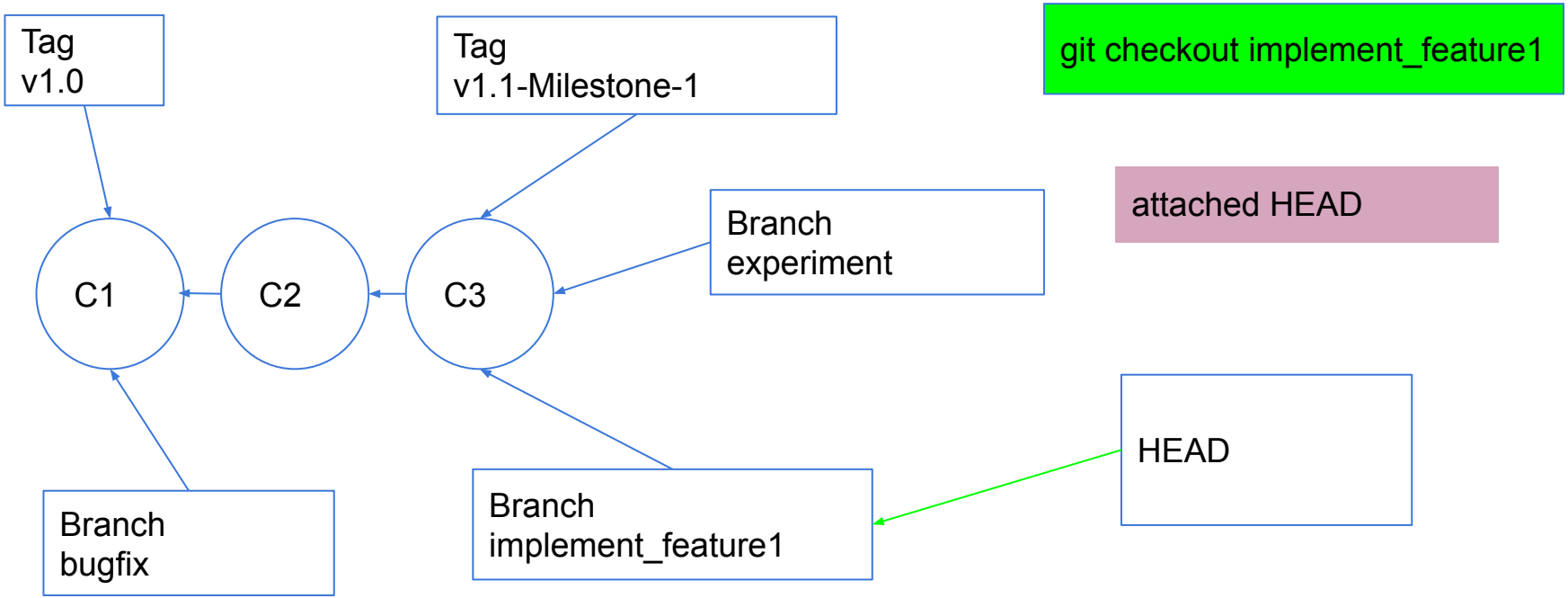
Bewegen des HEAD: git checkout <hash>



Bewegen des HEAD: git checkout <tag>



Bewegen des HEAD: git checkout <branch>



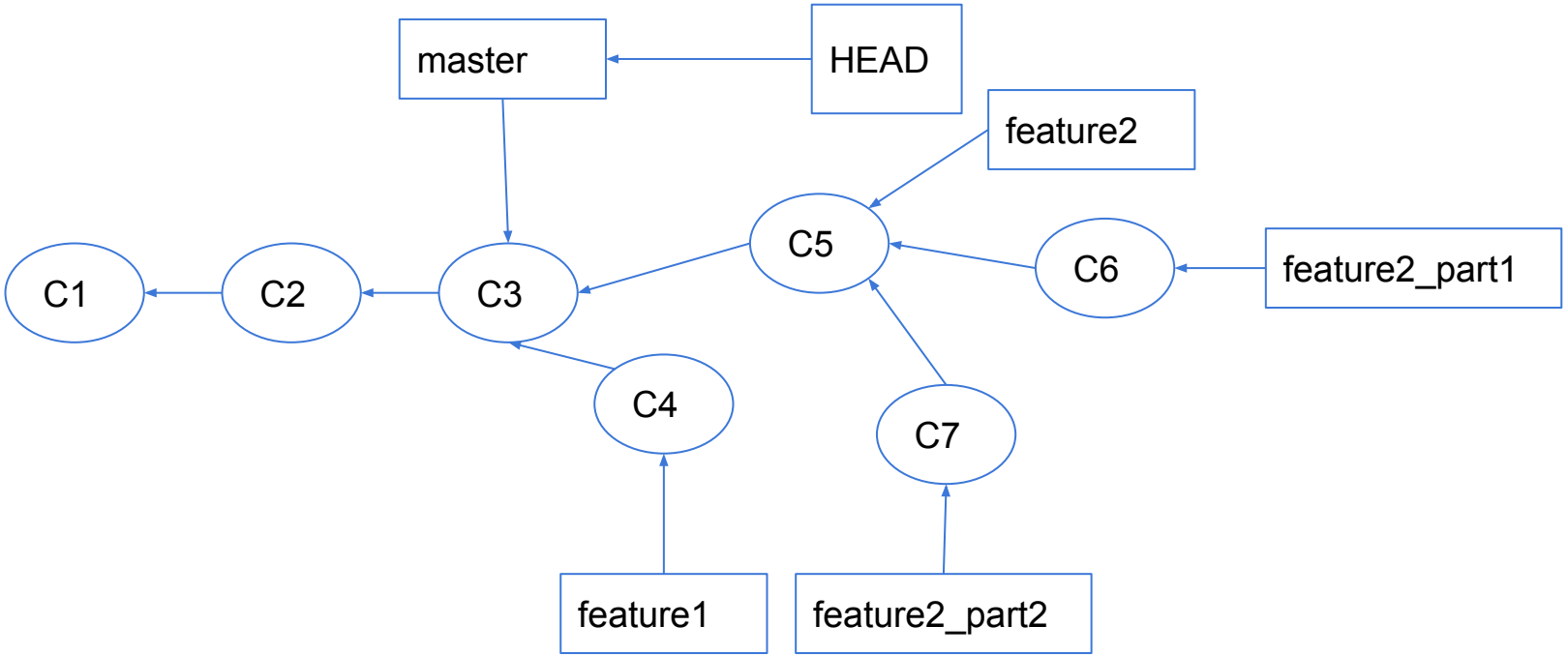
Bemerkung zum checkout

- Ein checkout ändert den Zustand des Workspaces
- Best Practice
  - Checkout nur in unauffälligem Status
  - Später: Wie arbeite ich bei einem auffälligem Status
    - checkout -f
    - reset
    - Stashing
    - Work in Progress-Branch

Exkurs: Command-Aliase

- git log --oneline --decorate --all --graph
- git config --global alias.pl “log --oneline --decorate --all --graph”

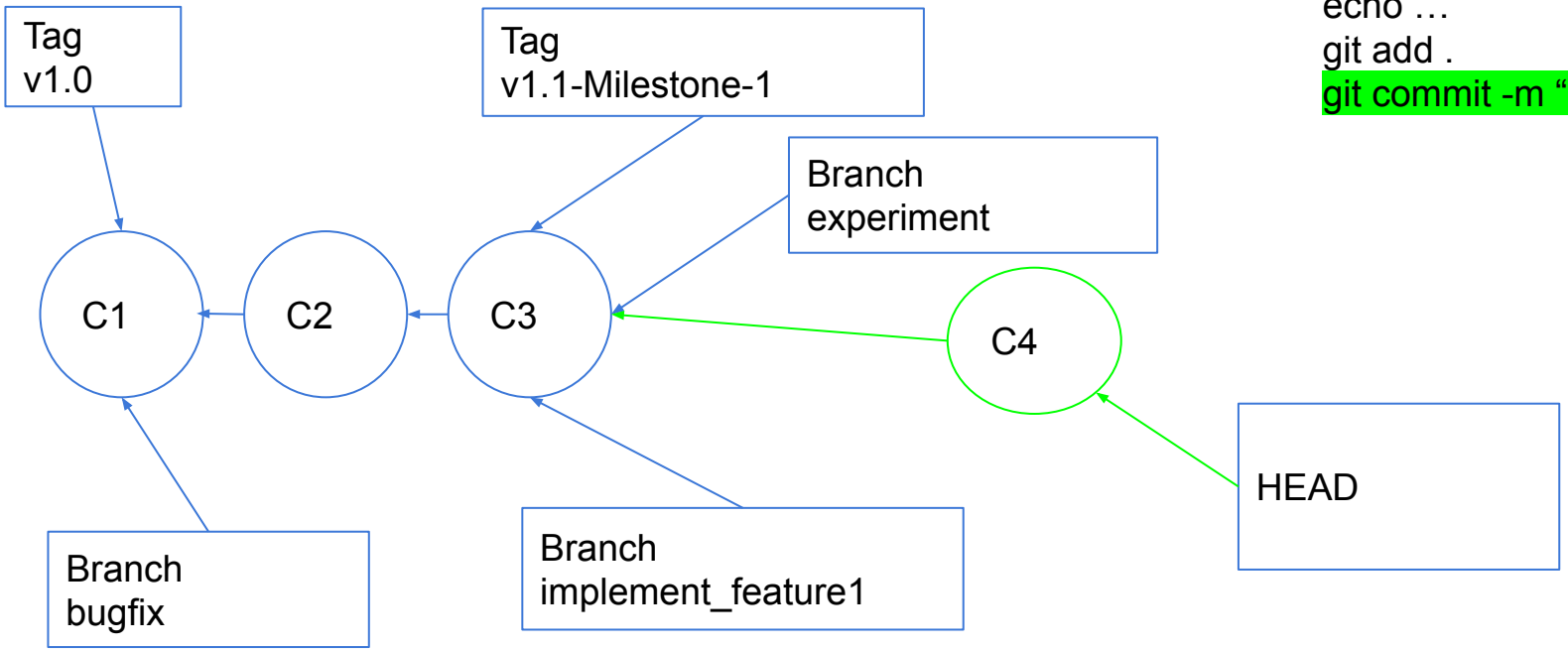
training branches



ToDo

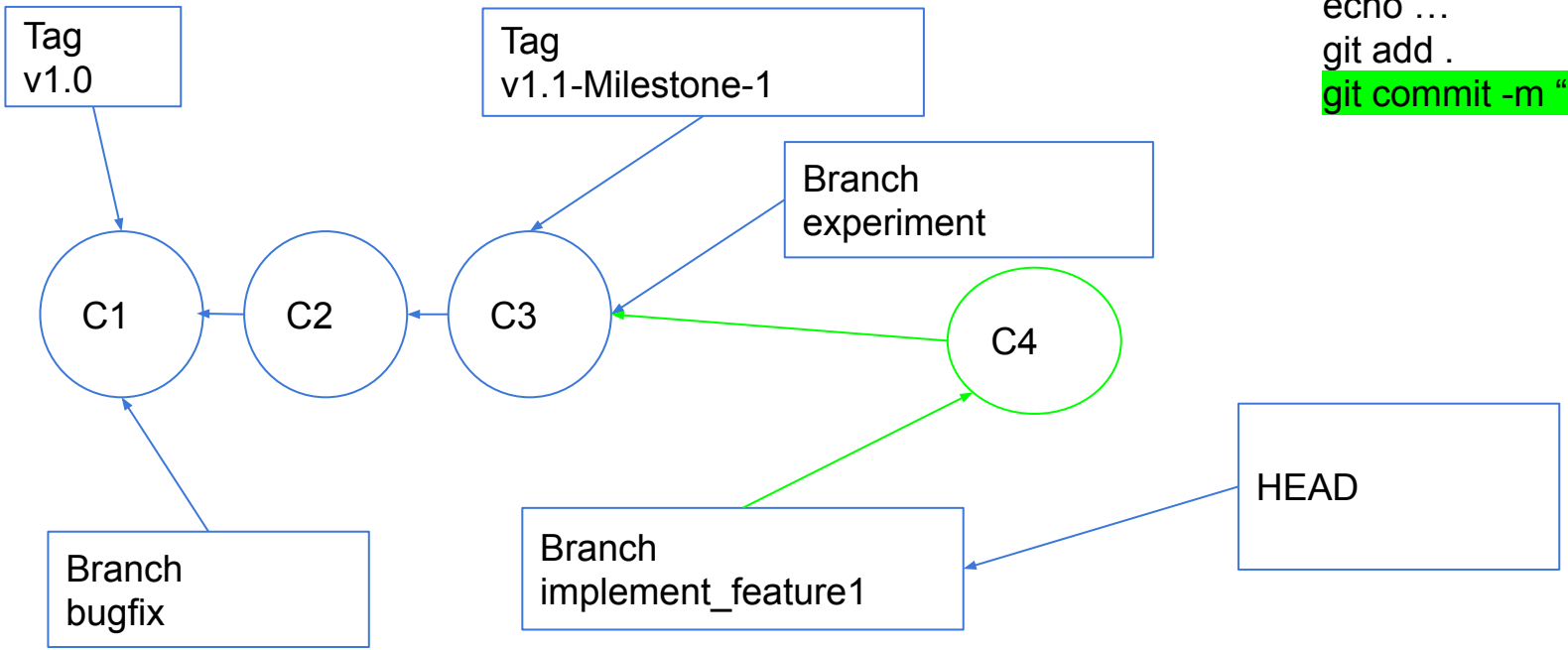
- Kopieren Sie das Skript, das Ihnen diese Struktur bereitstellt
- Visualisieren und Verifizieren Sie dieses Struktur durch den “pretty log”-Alias-Befehl
- checkout mit sauberem Status
  - Attached HEAD versus detached HEAD
    - Ausgabe des checkout-Befehls
    - git status
- Weitere Branches und Tags anlegen
  - git branch new_name
    - Vorsicht: Der HEAD ist nicht an diesen neu erzeugter Branch attached!
  - git checkout -b new_branch <hash> | <tag> | <branch>
- Branches, die nicht alleine an der Spitze einer Reihe stehen, können gelöscht werden: git branch -d <name>

git commit: Detached HEAD



echo …
git add .
git commit -m “...”

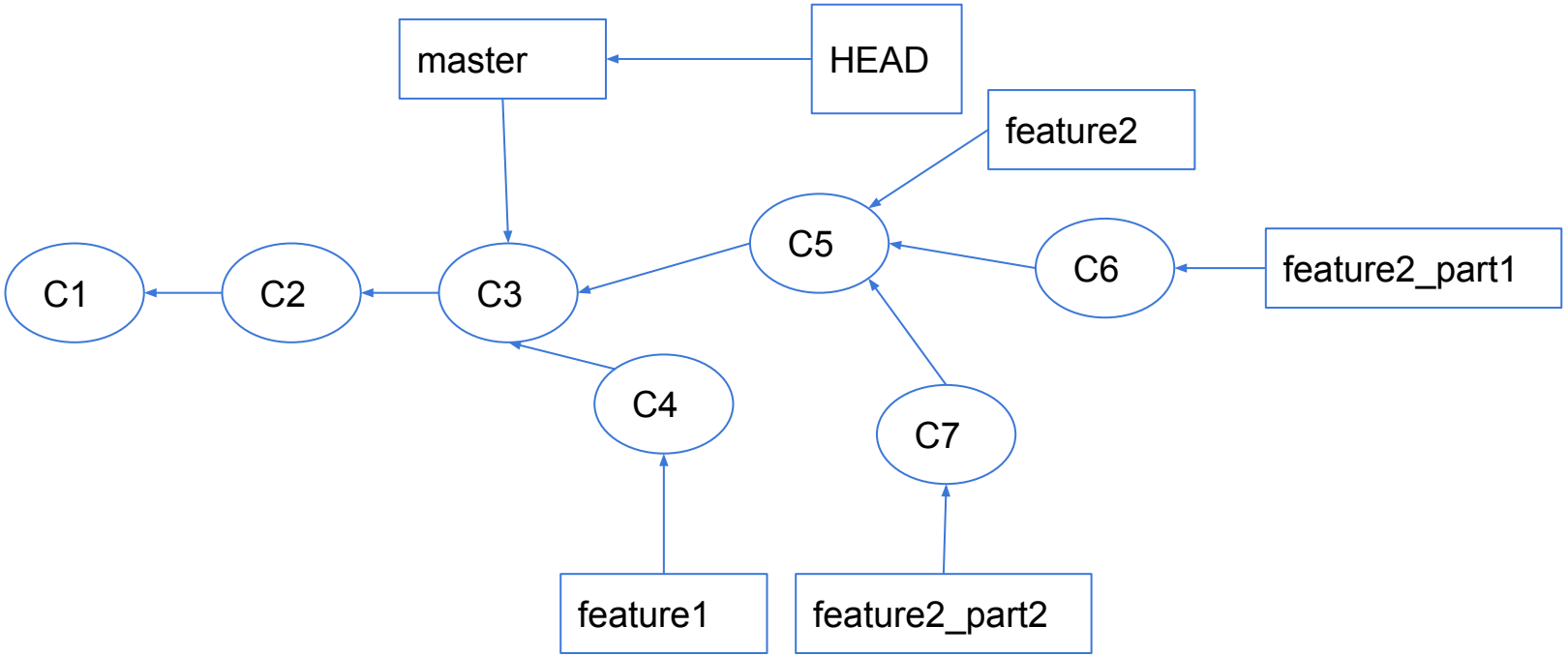
git commit: Attached HEAD




```
echo …
git add .
git commit -m "..."
```

Konsolidieren von Branches

Ausgangssituation

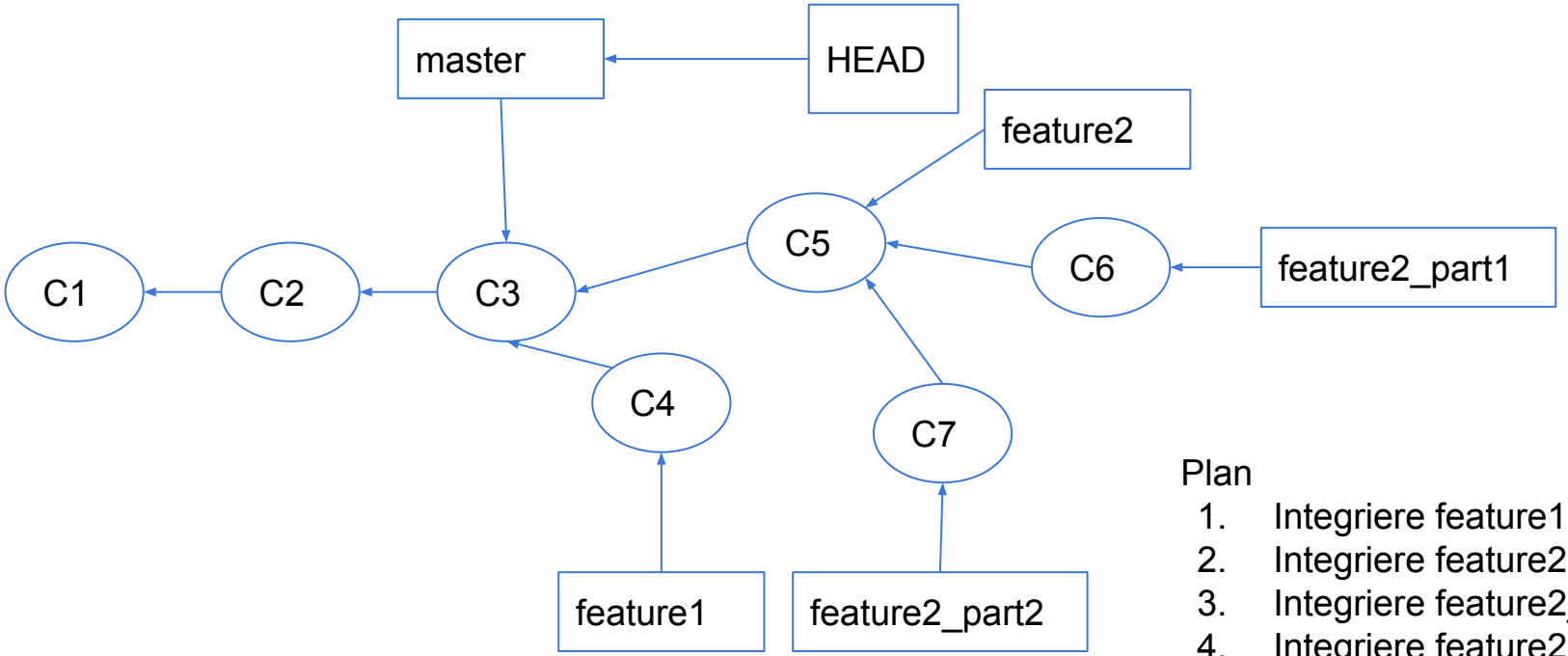


Zieldefinition

- “Alle Änderungen, die in den feature-Branches eingeführt wurden, sollen in den master überführt werden”
  - Fachlich: Wir haben die features fertig entwickelt und wollen nun unseren Software-Stand weiterentwickeln
- Historie soll exakt die durchgeführten Änderungen wiederspiegeln

Strategie: Merging

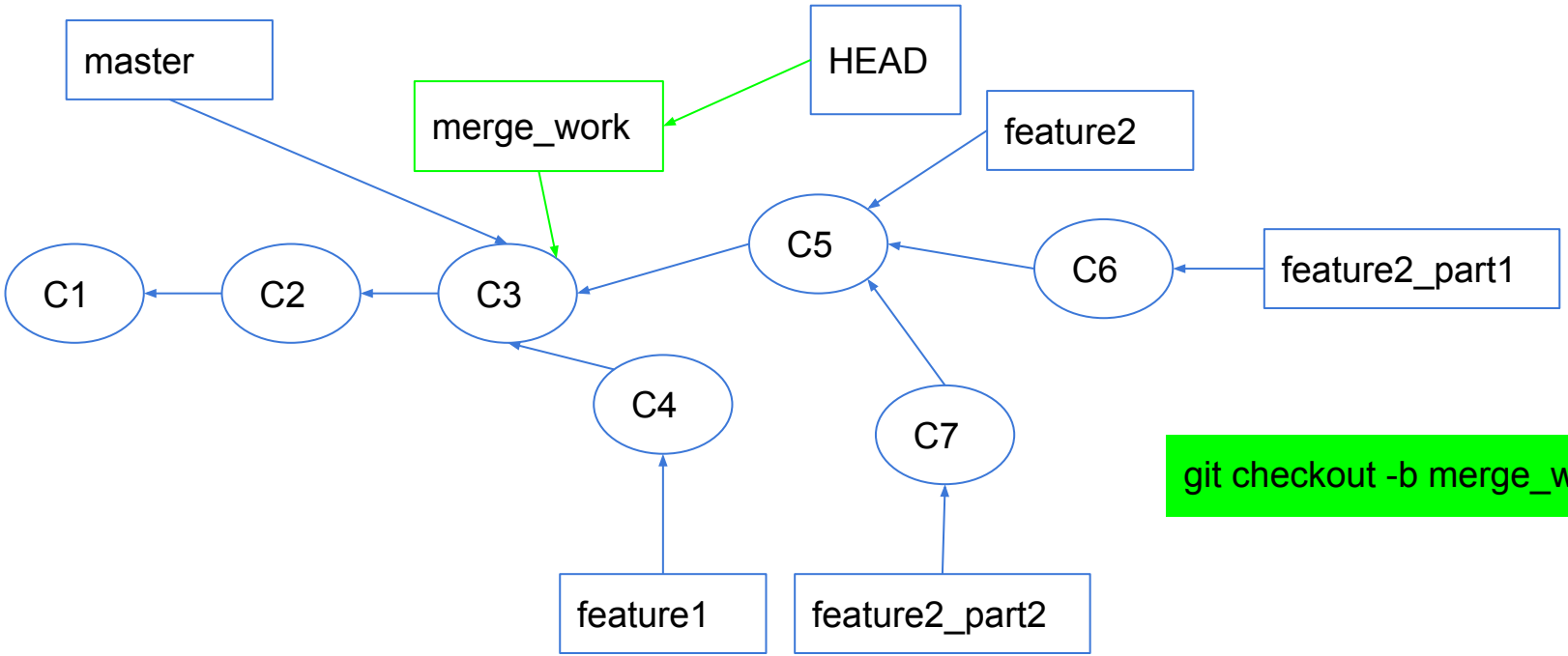
Merge Plan



Plan

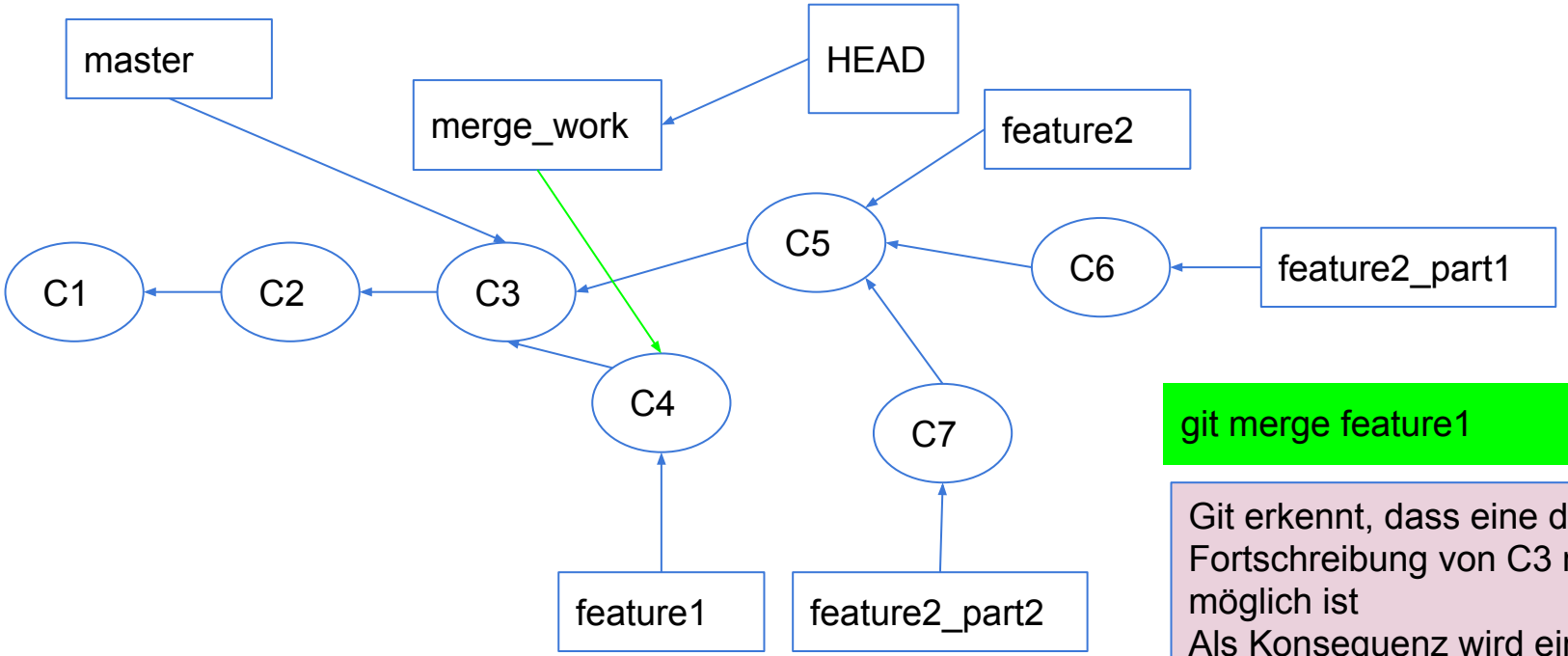
1. Integriere feature1
2. Integriere feature2
3. Integriere feature2_part1
4. Integriere feature2_part2

Arbeiten Sie vorsichtig!




```
git checkout -b merge_work
```

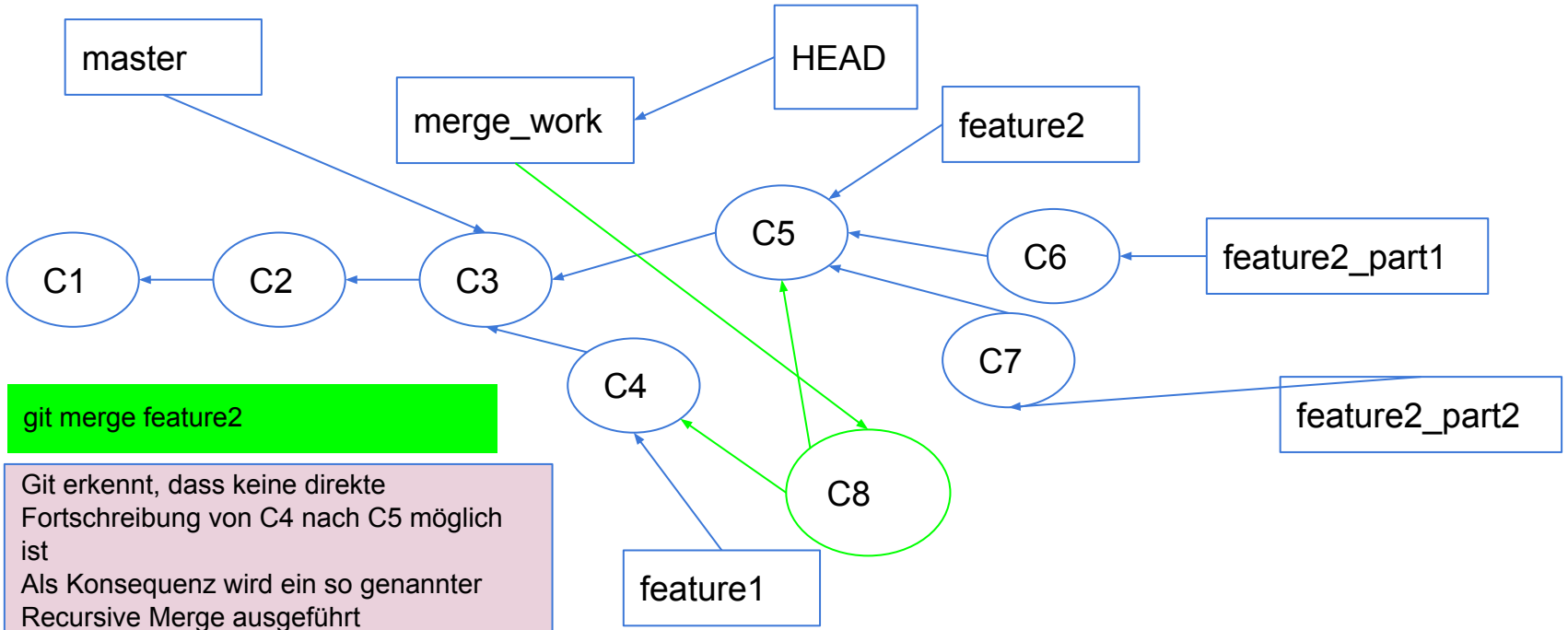
Schritt 1: Integriere feature1 in merge_work



git merge feature1

Git erkennt, dass eine direkte Fortschreibung von C3 nach C4 möglich ist
Als Konsequenz wird ein so genannter Fast Forward ausgeführt
Merge-Konflikte durch inkonsistente Änderungen sind unmöglich!

Schritt 2: Integriere feature2 in merge_work



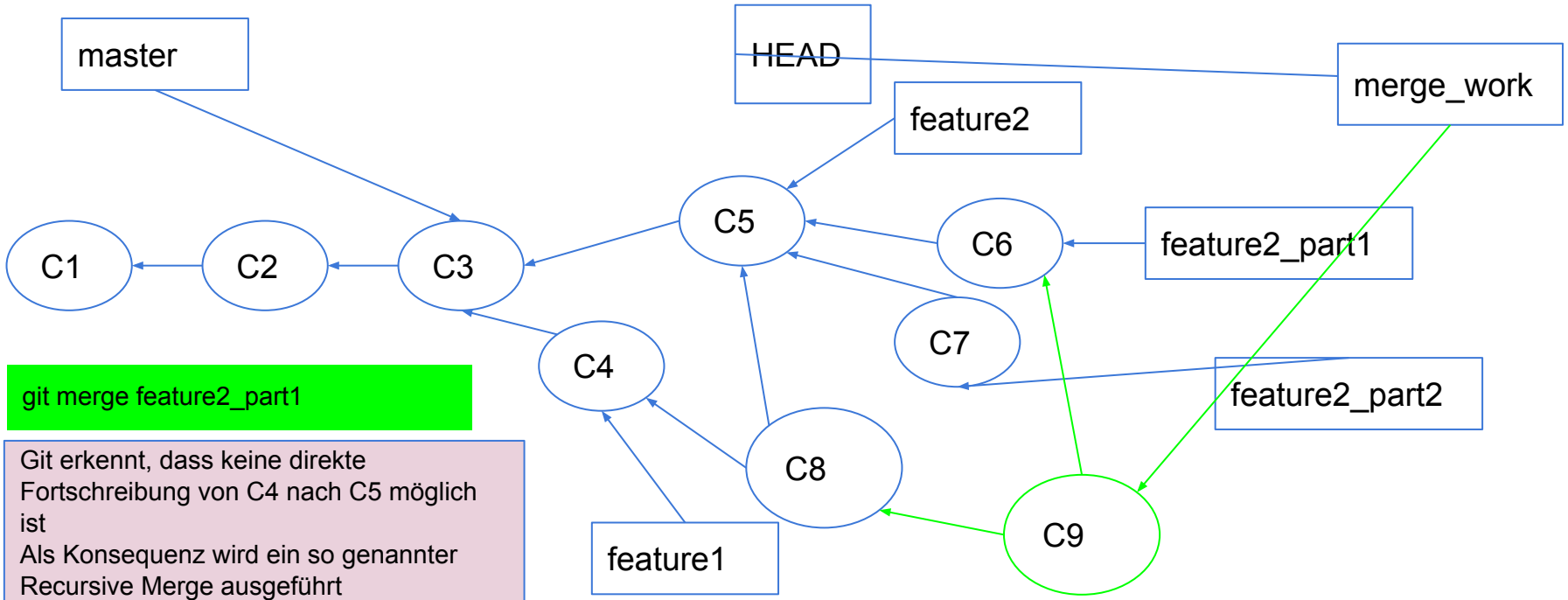
git merge feature2

Git erkennt, dass keine direkte Fortschreibung von C4 nach C5 möglich ist
Als Konsequenz wird ein so genannter Recursive Merge ausgeführt
Merge-Konflikte durch inkonsistente Änderungen sind immer möglich!
Git erkennt solche Konflikte sicher + beinhaltet eine Auto Resolution

Exkurs: Auto-Resolution

- Im Standard-Merging werden
  - Änderungen an unterschiedlichen Dateien
  - In unterschiedlichen Zeilen derselben Datei
- nicht als Konflikt signalisiert, sondern automatisch “behoben”
- VORSICHT
  - Das muss unbedingt im Rahmen des Mergens verifiziert/geprüft/getestet werden

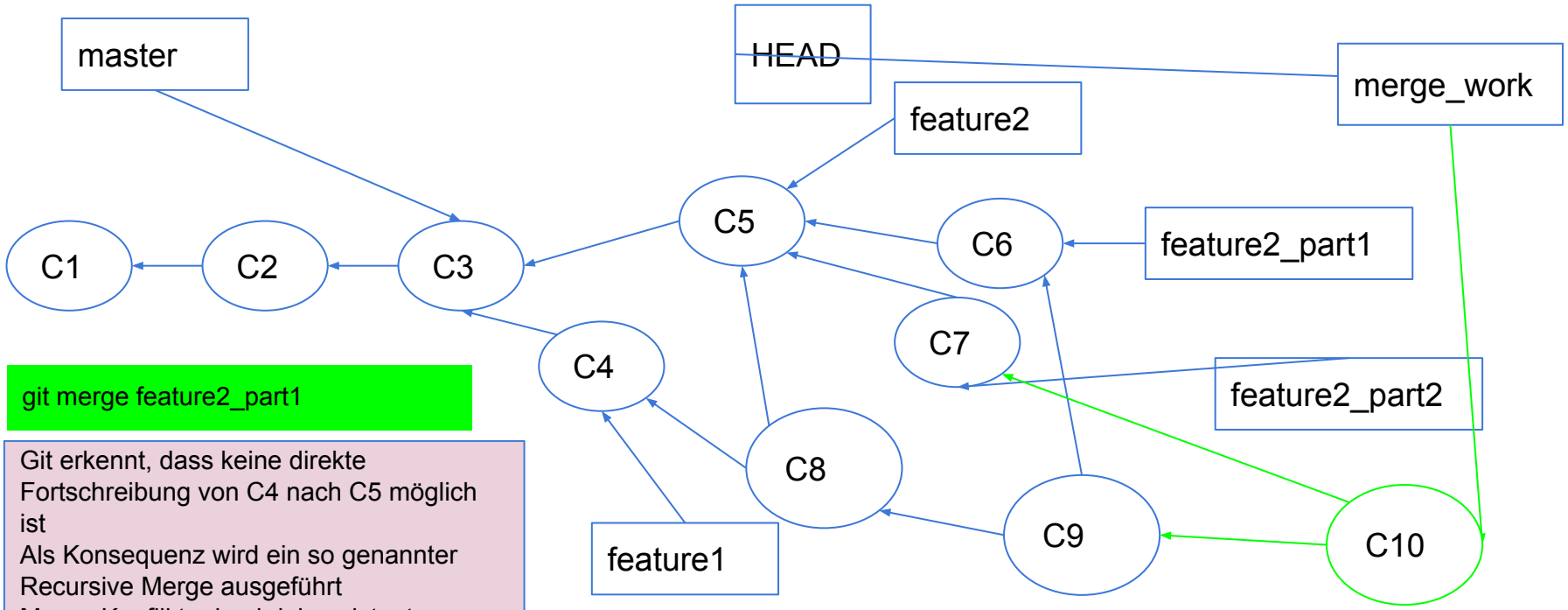
Schritt 3: Integriere feature2_part1 in merge_work



git merge feature2_part1

Git erkennt, dass keine direkte Fortschreibung von C4 nach C5 möglich ist
Als Konsequenz wird ein so genannter Recursive Merge ausgeführt
Merge-Konflikte durch inkonsistente Änderungen sind immer möglich!
Git erkennt solche Konflikte sicher + beinhaltet eine Auto Resolution

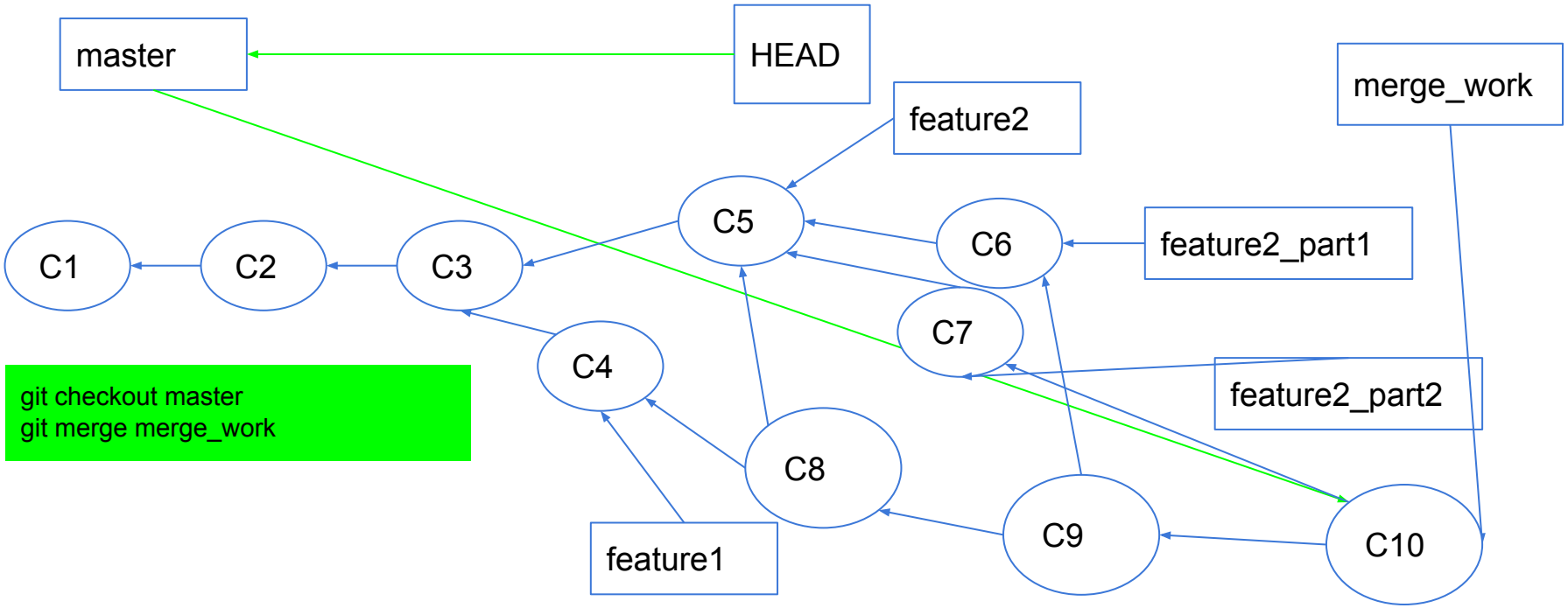
Schritt 4: Integriere feature2_part2 in merge_work



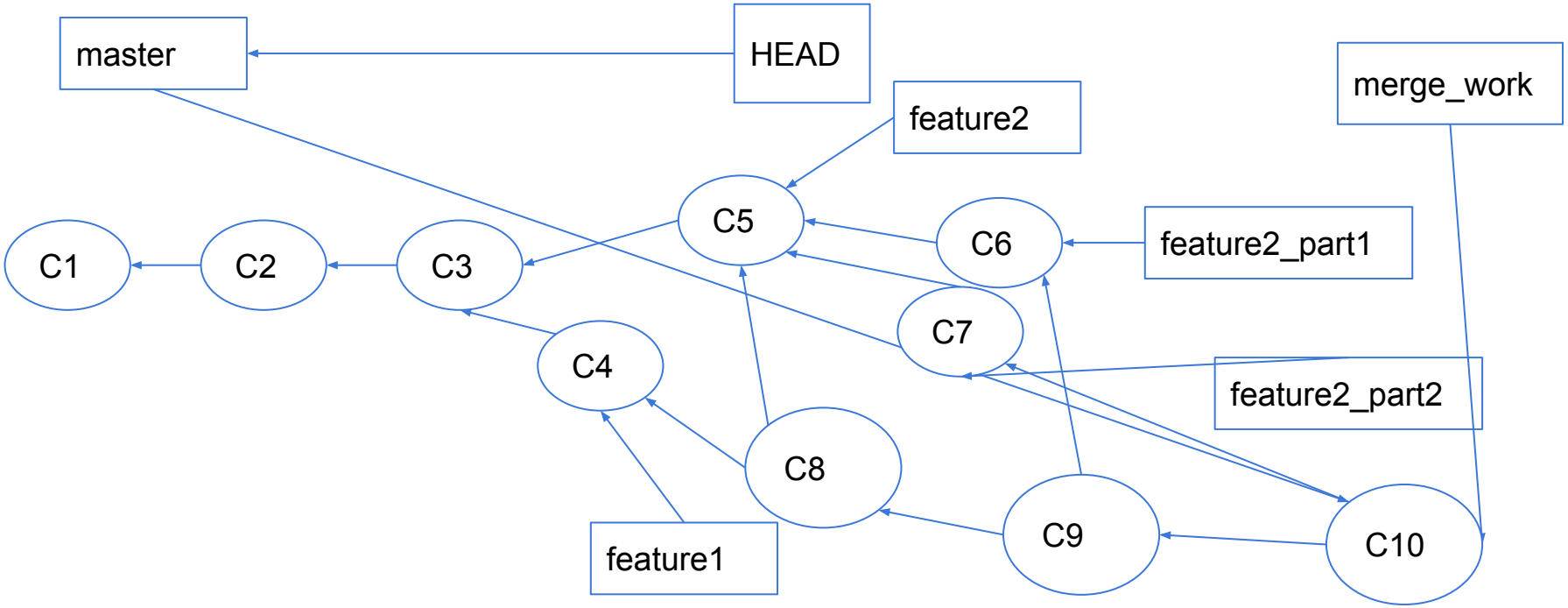
git merge feature2_part1

Git erkennt, dass keine direkte Fortschreibung von C4 nach C5 möglich ist
Als Konsequenz wird ein so genannter Recursive Merge ausgeführt
Merge-Konflikte durch inkonsistente Änderungen sind immer möglich!
Git erkennt solche Konflikte sicher + beinhaltet eine Auto Resolution

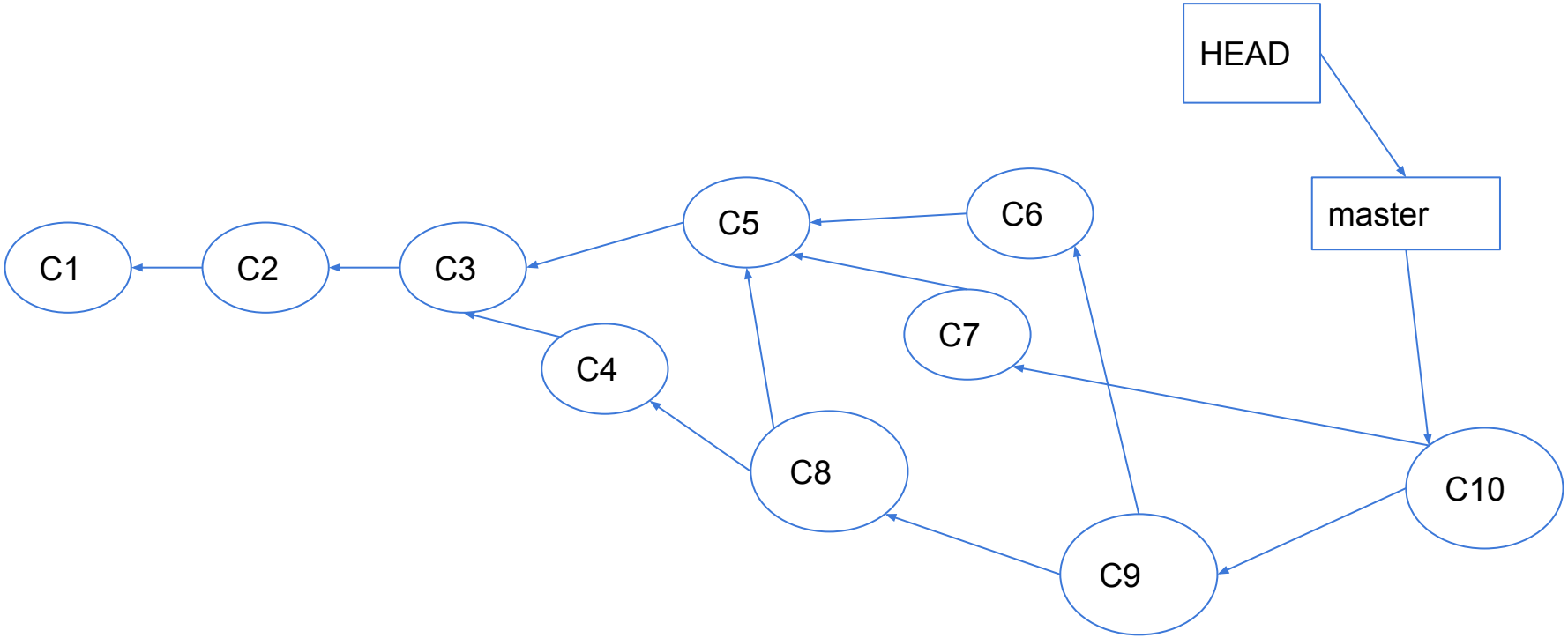
Schritt 5: Vorziehen des master



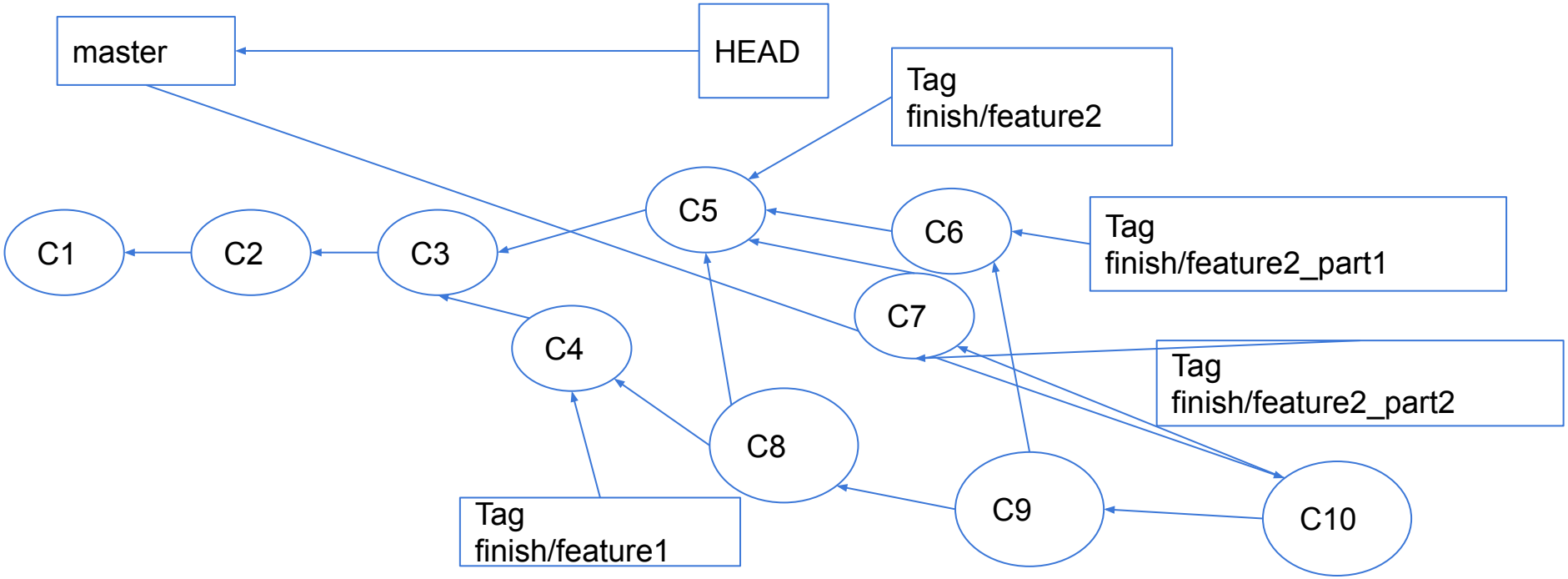
Schritt 6: Aufräumen (optional)



Schritt 6: Aufräumen (gnadenlos)



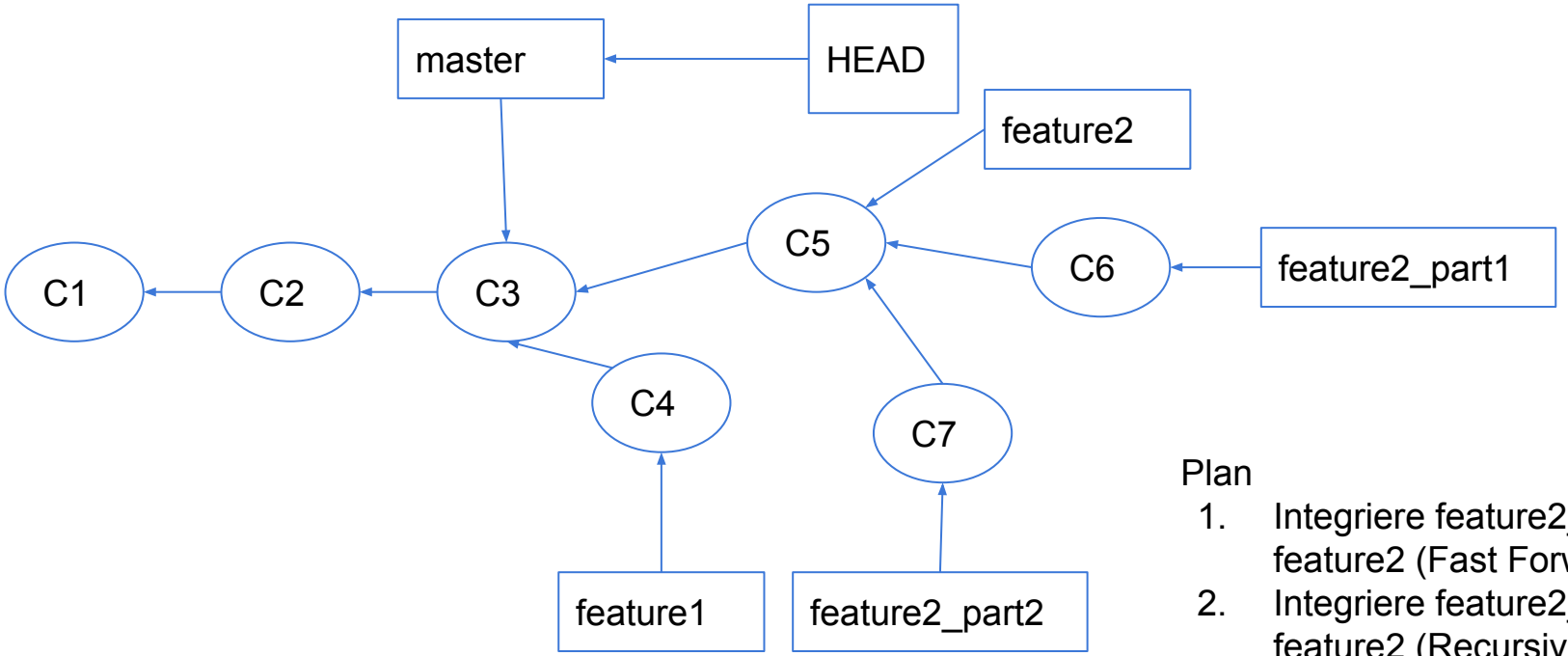
Schritt 6: Aufräumen



training_branches_merging_page47-58.zip


```
rainer@rainer-Aspire-VN7-572G:~/git_training/training_branches$ git pl
*   d9d19eb (HEAD -> master, merge_work) merge feature2_part2 and resolve confli
cts
|\
| * eade7be change content-feature2, part2
* |   9cf5480 Merge branch 'feature2_part1' into merge_work
|\ \
| * | 5934ead change content-feature2, part1
| |/
* | 1b439be Merge branch 'feature2' into merge_work
|\|
| * b3e53d0 add content-feature2
* | 6f6fa51 add content-feature1
|/
* 7a5f3ef change content
* 41074d1 add content
* 752bdf0 setup project
```

ToDo: Merge Plan



Plan

1. Integriere feature2_part1 in feature2 (Fast Forward)
2. Integriere feature2_part2 in feature2 (Recursive MIT conflict)
3. Integriere feature1 in feature2 (Recursive mit Auto Resolution)

Unsauberer Status

Entstehung

- Änderungen im Workspace, die noch nicht dem Repository hinzugefügt wurden
  - Bisher ist das ganz normales Arbeiten
- Nun: “Notfall”: Es ist notwendig, sofort an einem Stand weiterzuarbeiten

git reset

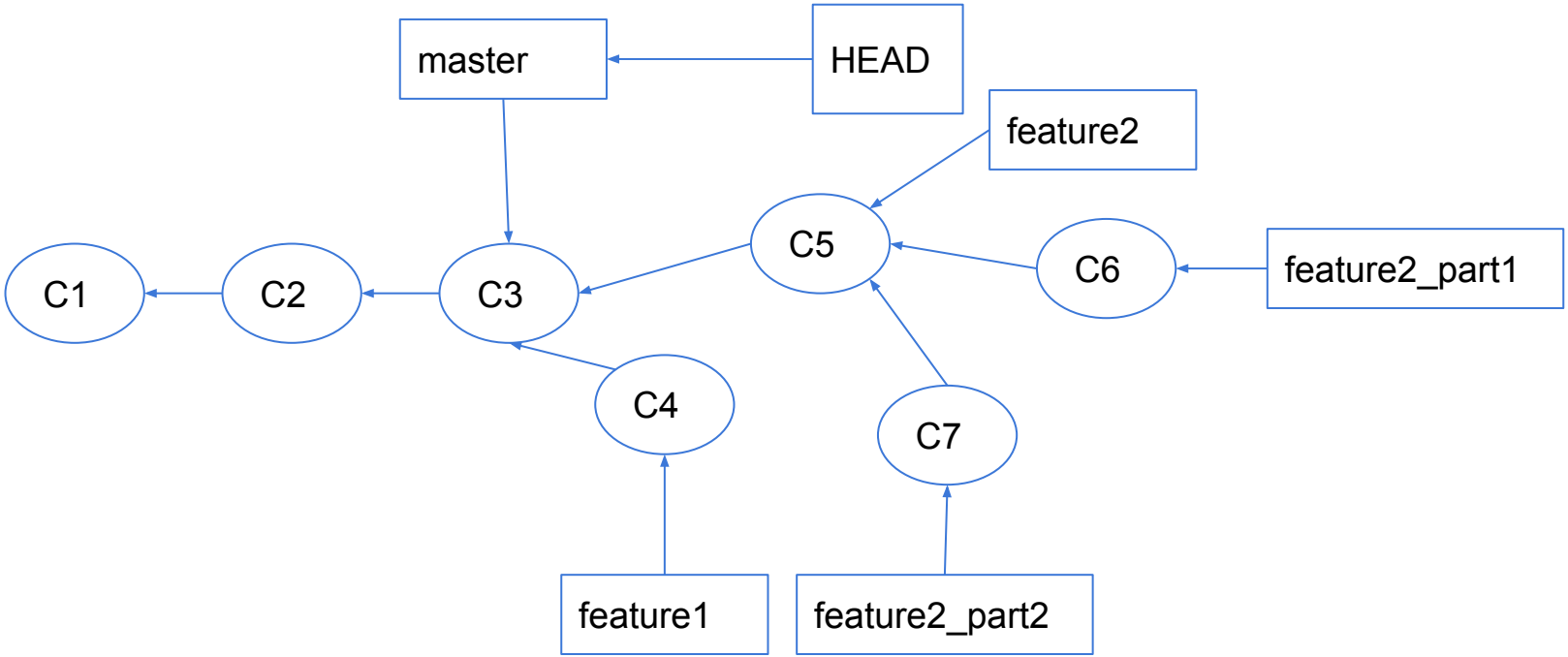
- Bewegt einen Branch “zurück”
- Modi für die Synchronisation mit dem Workspace
  - --hard
  - --soft
- Für die eben beschriebene Problemstellung keine Lösung

Möglichkeiten

- Arbeiten mit den bisher bekannten Befehlen
  - Anlegen eines WIP-Branches
  - Darin adden und committen
- Der Stash des Repositories
  - VORSICHT
    - Klar konzipiert für diese Notfall-Maßnahme
    - Stashes sind immer lokal
  - add . + stash

Konsolidieren von Ständen, Teil 2

Ausgangssituation

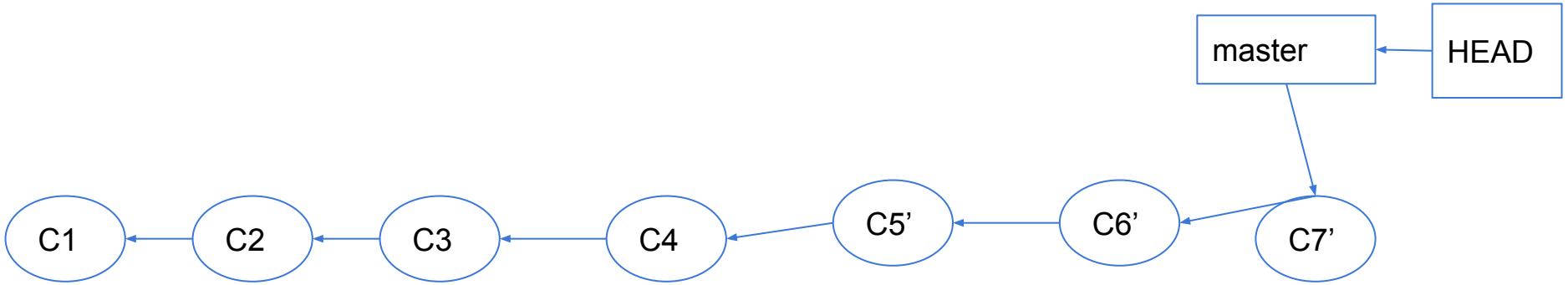


Zieldefinition

- “Alle Änderungen, die in den feature-Branches eingeführt wurden, sollen in den master überführt werden”
  - Fachlich: Wir haben die features fertig entwickelt und wollen nun unseren Software-Stand weiterentwickeln
- Historie soll eine stringente Dokumentation darstellen

Strategie: Rebase

Ergebnis



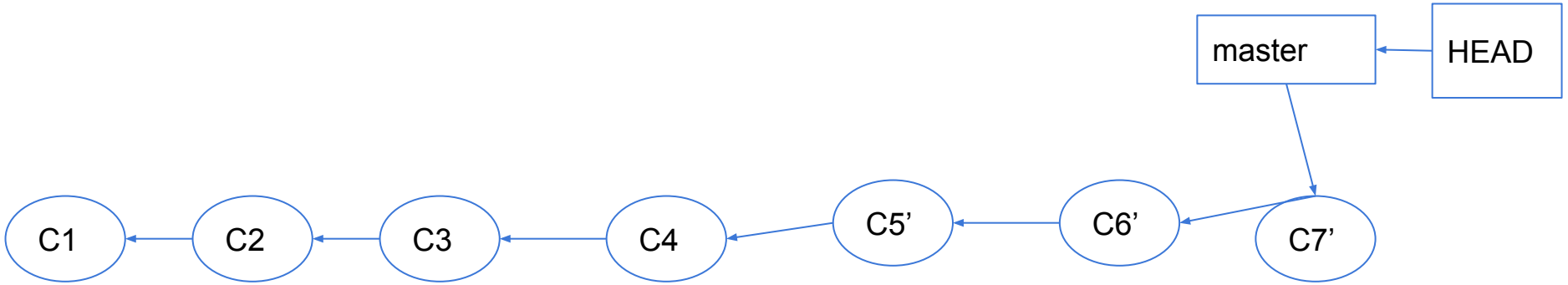
Rebasing: Schritt für Schritt

- git checkout feature2_part1
- git rebase feature1
  - AUCH HIER KÖNNEN KONFLIKTE AUFTRETEN!!!
  - Konflikte können beim Rebasen mehrfach hintereinander auftreten
    - Nach einer Konfliktlösung add . + rebase --continue
      - git rebase --abort
      - git rebase | merge --dry-run
- git checkout feature2_part2
- git rebase feature2_part1
- Vorziehen des Masters
  - git checkout master + git merge feature2_part2
- Aufräumen

ToDo

- Vollziehen das Rebasing-Beispiel nach
- Variieren Sie durch unterschiedliche Rebase-Plänen
- Fragen:
  - Wird durch Rebasing etwas vorhandenes geändert?
  - Werden Informationen gelöscht?
  - Wie können Sie an scheinbar gelöschte Informationen wieder rankommen?

Ausgangssituation

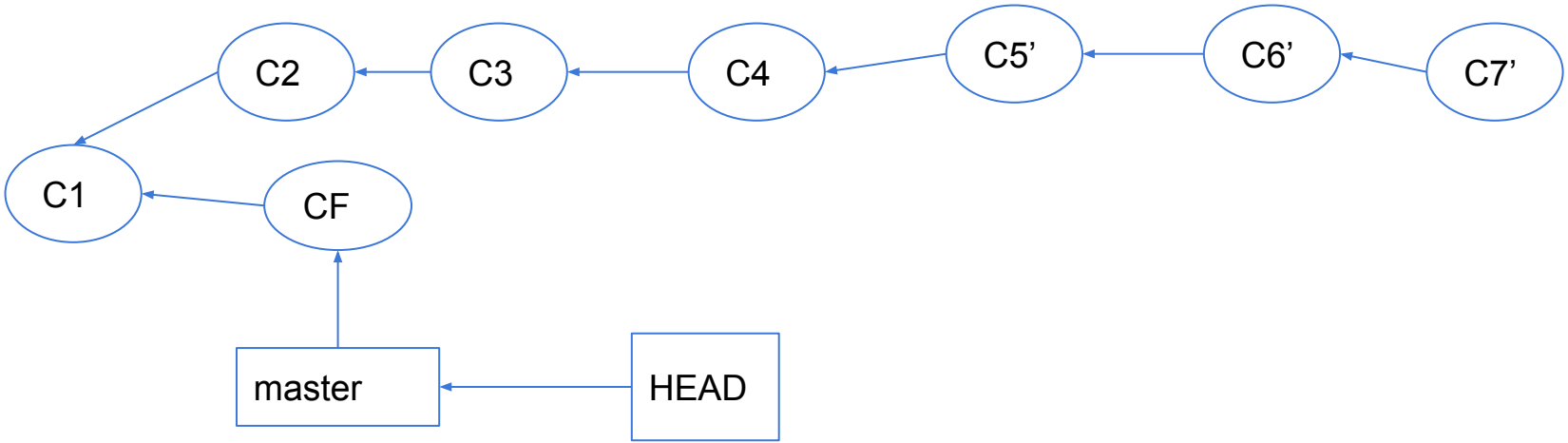


Zieldefinition

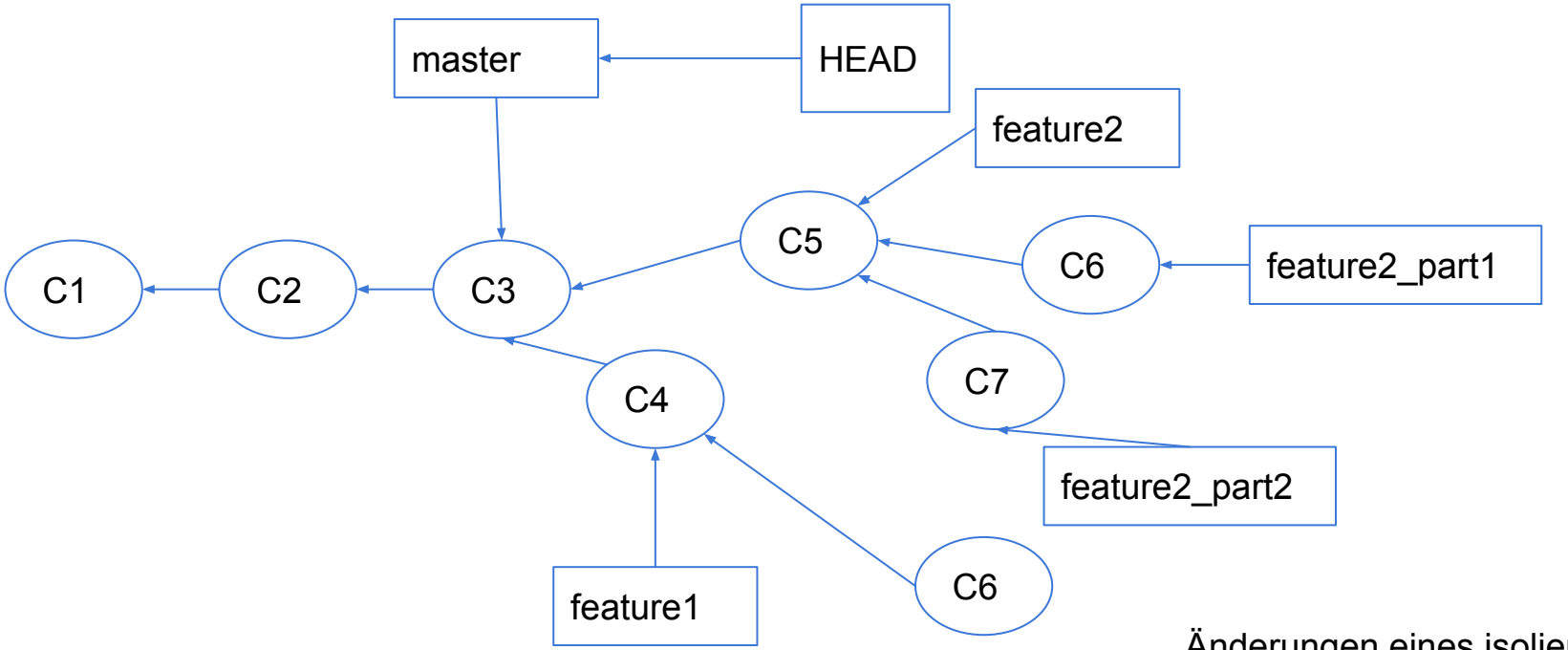
- Historie soll eine aussagekräftige, kompakte Dokumentation darstellen

Strategie: Interactive Rebase

Ausgangssituation



cherry-pick



Änderungen eines isolierten Commits werden übernommen

Hinweise zum cherry-pick

- Paradebeispiel
  - Änderungen eines BugFixes werden an verschiedenen anderen Stellen eingespielt
- Die Git-Community empfiehlt den Einsatz von cherry-pick aktuell nicht mehr
  - Meistens ist ein Merge oder Rebase geeigneter als ein cherry-pick
  - git checkout <branch|tag|hash> <path>

Best Practices im Arbeiten mit Git = “Git Flows”

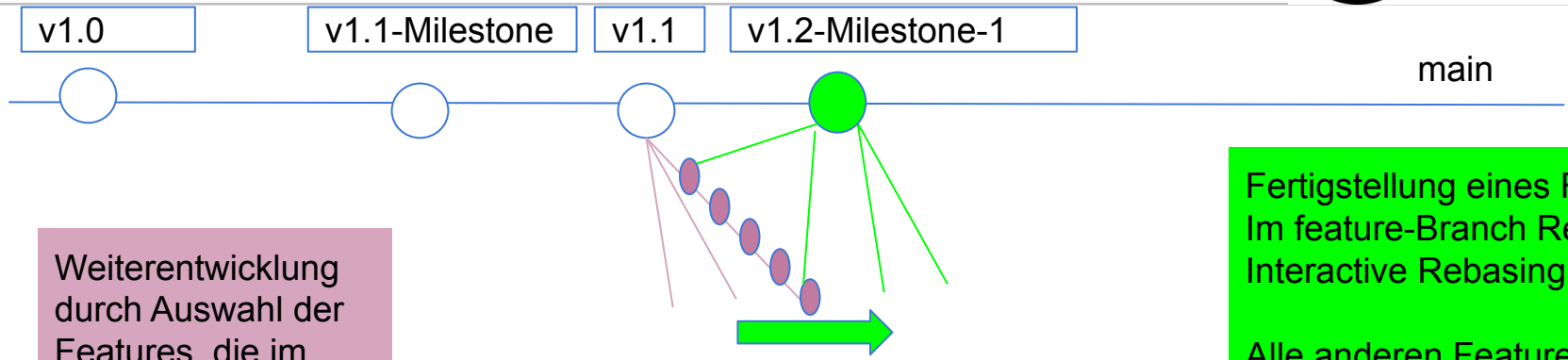
Allgemeiner Konsens

- Es gibt zumindest einen langlebigen Branch
  - “master” oder “main”
- Jegliche Änderung an einem Software-Projekt wird ausschließlich in einem Feature-Branch gemacht, nie am master-Branch
  - Neues Feature
  - BugFix
  - Enhancement
  - …
- Sinnvoll sind Namenskonventionen
  - feature/my_new_feature
  - fix/my_bug_fix
  - ...

2 Standard-Flows

- GitHub-Flow
  - Stammt aus der GitHub-Community und ist trotzdem unabhängig vom GitHub-Server
- GitFlow
  - Atlassian-Community
- Hinweis
  - Beide Flows sind nicht ohn Anpassungen sinnvoll in eigenen Projekten einzusetzen
  - Sie sind eine gute Ausgangsbasis

GitHub-Flow

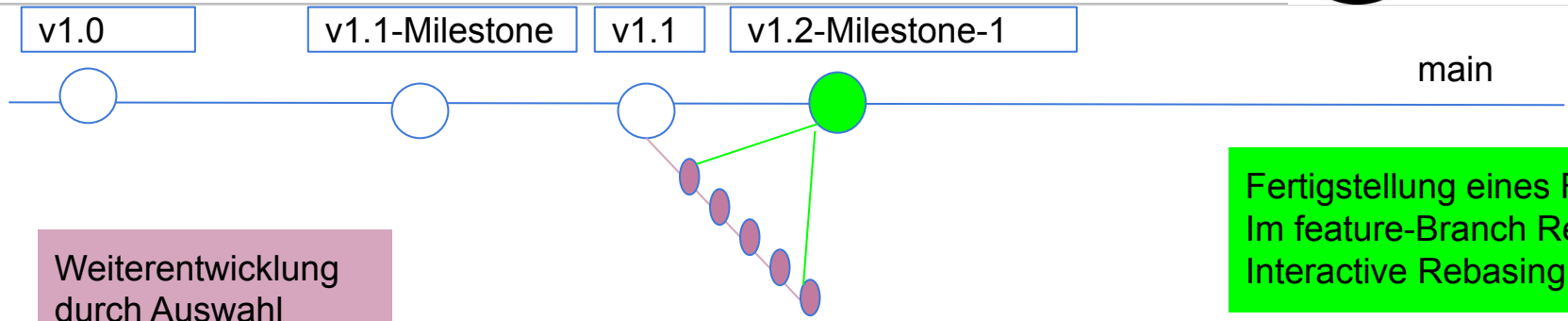


Weiterentwicklung durch Auswahl der Features, die im nächsten Release bereitgestellt werden sollen

Fertigstellung eines Features
Im feature-Branch Rebasing und Interactive Rebasing

Alle anderen Features werden auf diesen Milestone rebased

GitHub-Flow: Single Developer



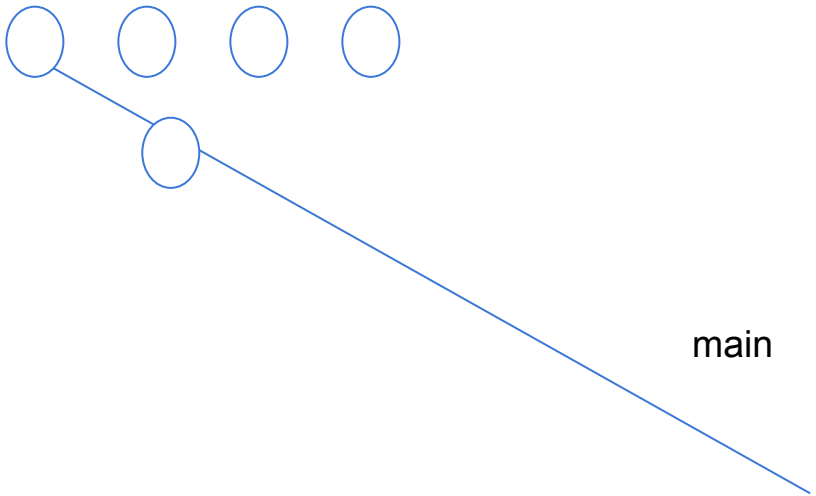
Weiterentwicklung durch Auswahl eines Feature

feature Branche

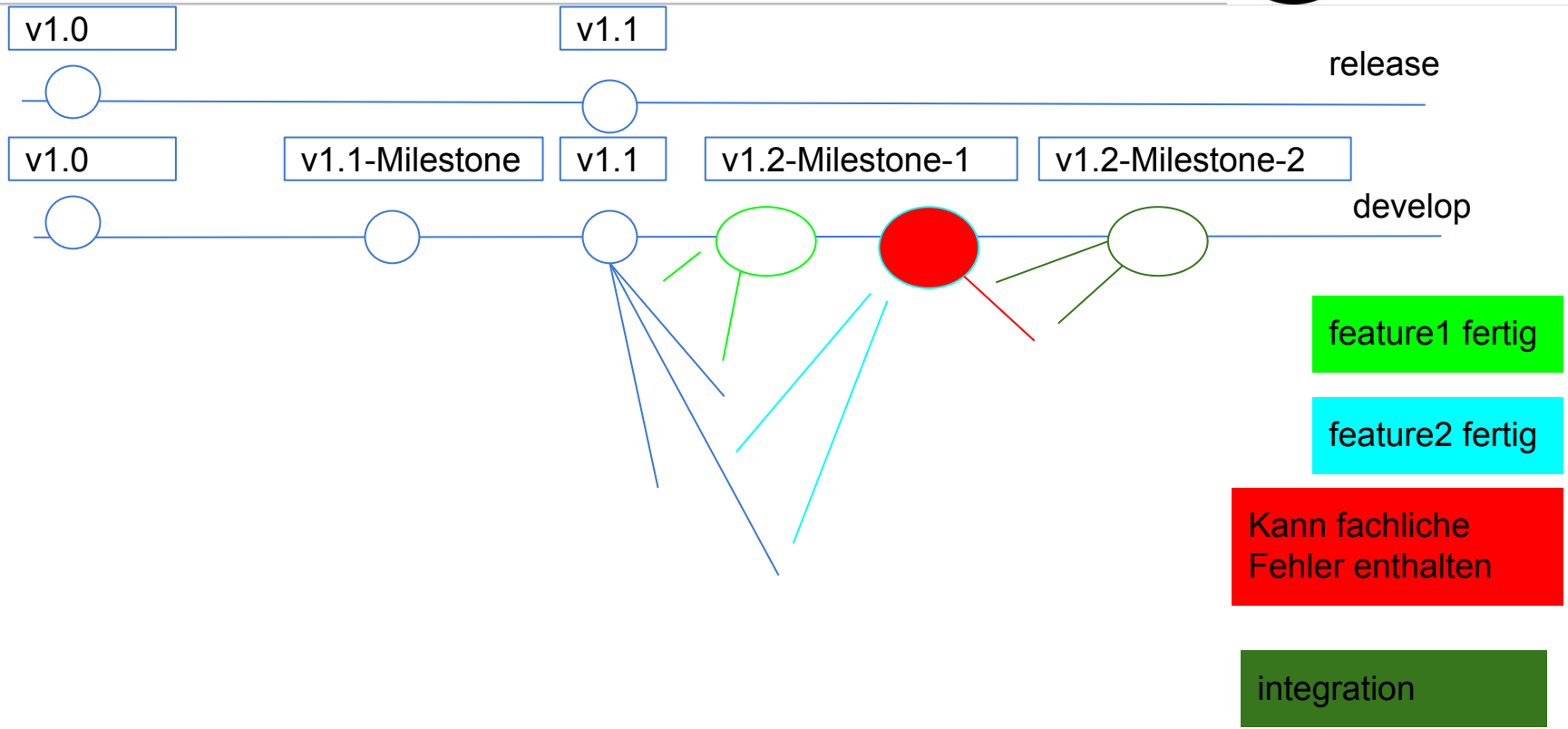
Fertigstellung eines Features Im feature-Branch Rebasing und Interactive Rebasing

Zum Rebasing und Interactive Rebasing

- Vorsicht
  - Ein Rebasing des masters/main führt zu Chaos



GitFlow


v1.0
v1.1
release
v1.0
v1.1-Milestone
v1.1
v1.2-Milestone-1
v1.2-Milestone-2
develop
feature1 fertig
feature2 fertig
Kann fachliche Fehler enthalten
integration

ToDo: Arbeiten mit GitHub Flow in der Single Developer-Variante

- Neues Projektverzeichnis
- Initiale Stand: Workspace mit einer Readme.txt
- Feature: Planet einführen
  - planet.txt
    - Venu
    - Venus
    - Mars
    - Saturn
- Feature: Stern
  - star.txt
    - Sol
    - Vega
    - Beteigeuze

Ziel:
Sauberer Master-Branch mit den beiden Commits “planet fertig”, “stern fertig”

Tagging
Milestones/Releases im Master
Fertiggestellten Features

Musterlösung

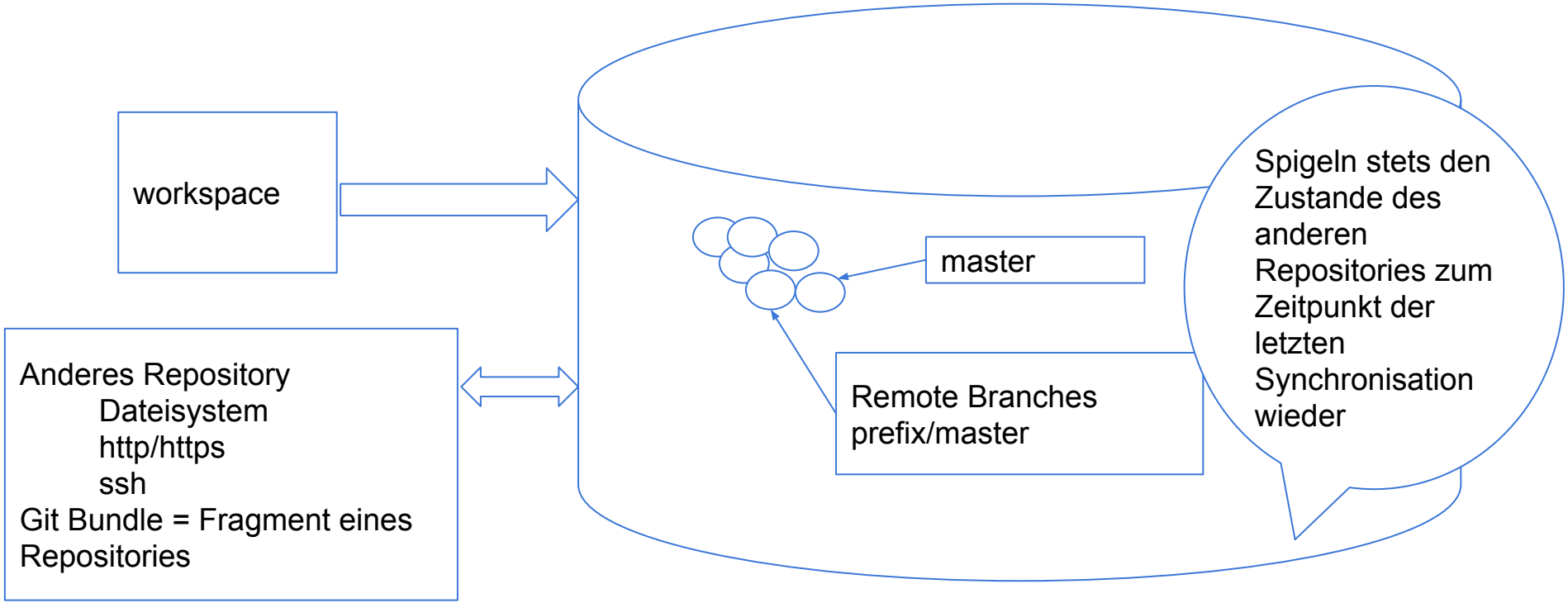

```
* b564edd (HEAD -> master, tag: v1.0) implement stars
| * 302bc92 (tag: finish_star) working on stars
| * c2b8263 working on stars
| * 2ddf416 working on stars
|/
* 4be21d9 (tag: v1.0-Milestone-1) implement planets
| * 2c02075 (tag: finish_planet) working on planets
| * dc71565 working on planets
| * ad05fbb working on planets
| * cba4875 working on planets
|/
* 6a94585 (tag: initial) initial
```

Auf GitHub

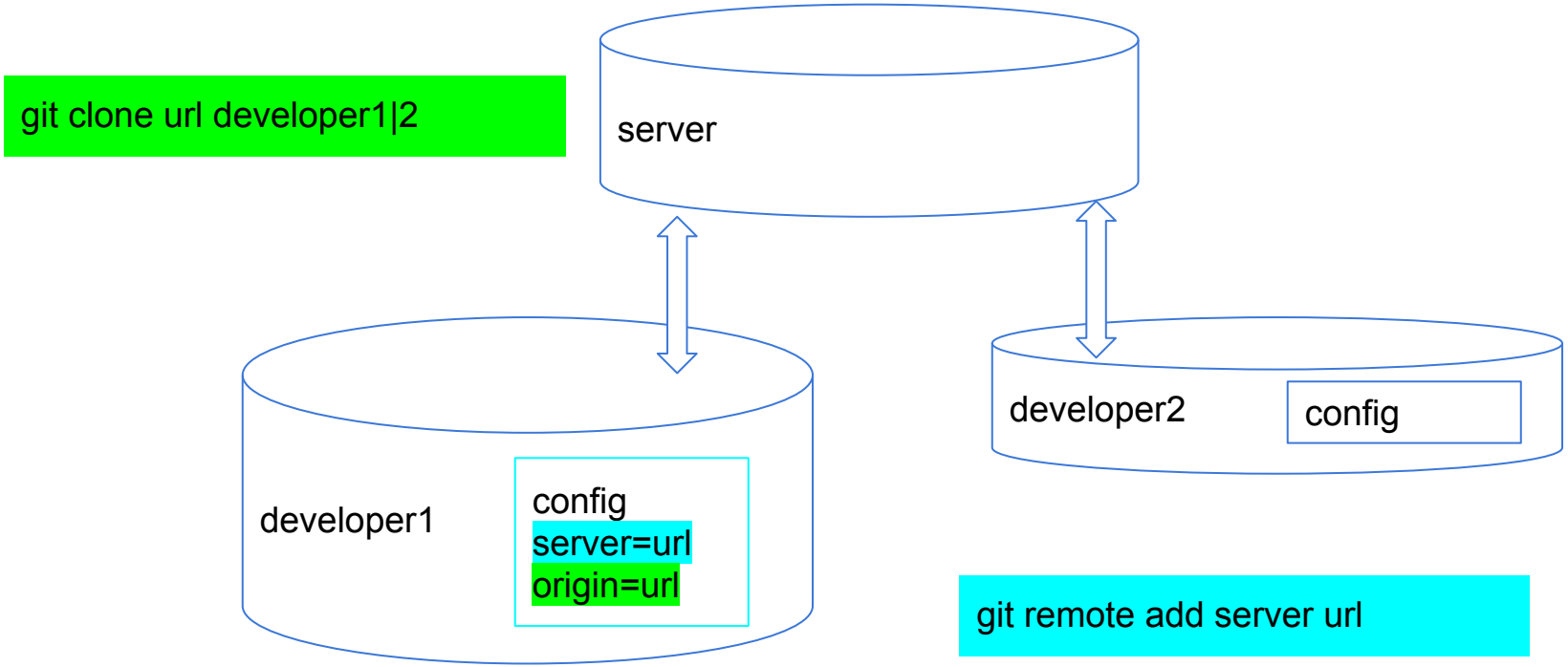
- Repo demo_github_flow.zip
- Befehle demo_github_flow.sh

Verteilte Repositories

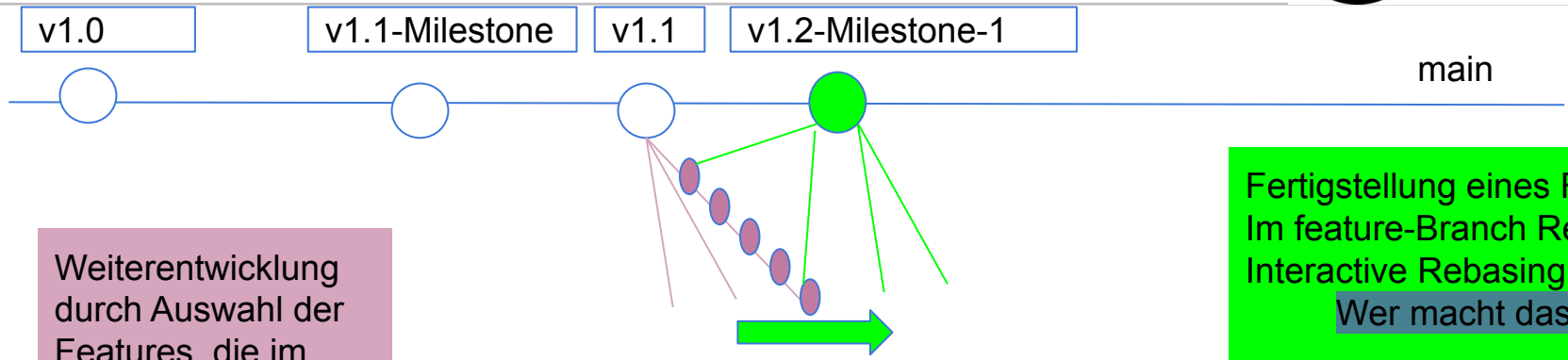
Grundprinzip: Remote Branches



Mehrere Developer, ein Server



GitHub-Flow: Revisited



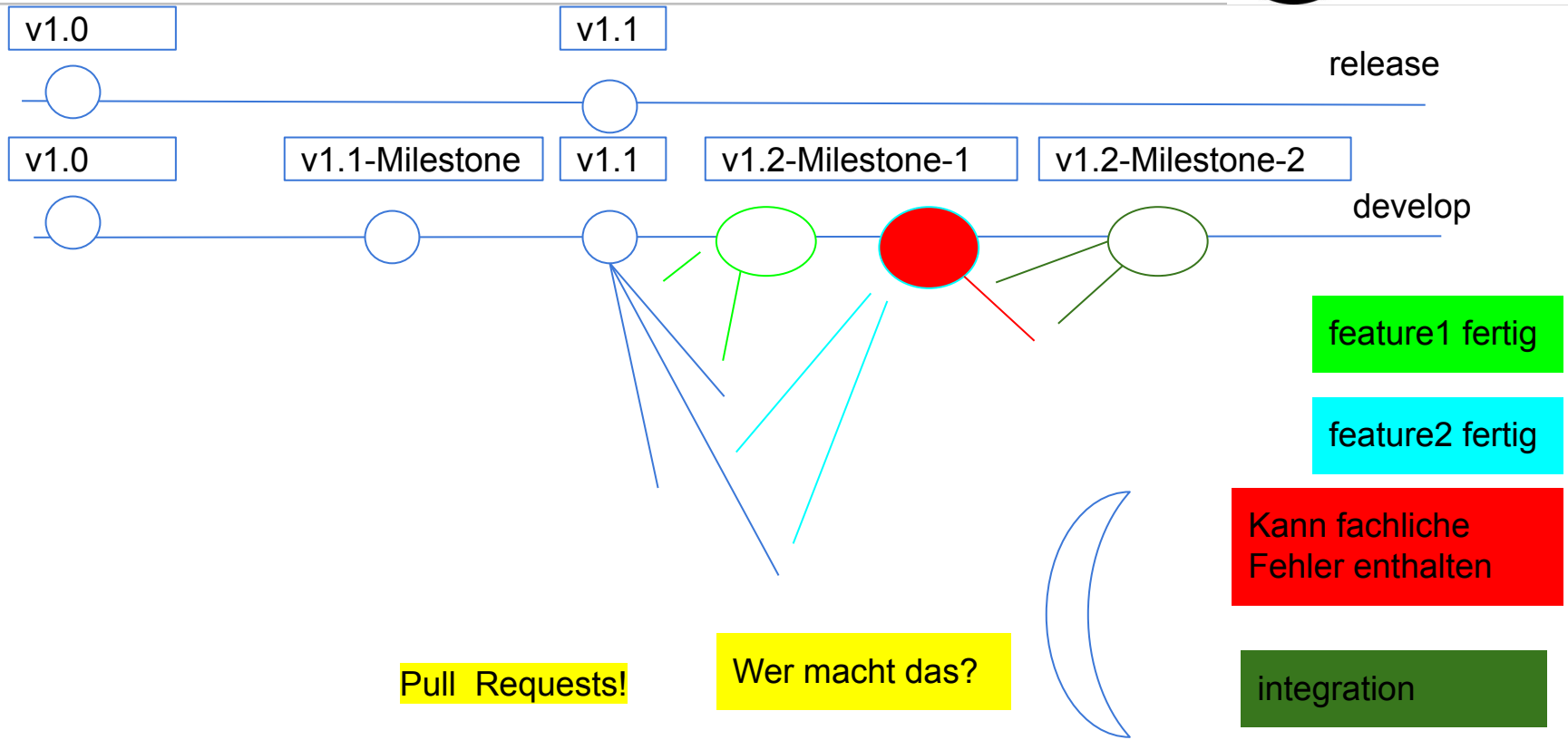
Weiterentwicklung durch Auswahl der Features, die im nächsten Release bereitgestellt werden sollen

Fertigstellung eines Features
Im feature-Branch Rebasing und Interactive Rebasing
Wer macht das?

Alle anderen Features werden auf diesen Milestone rebased
Wer informiert?

Pull Requests!

GitFlow: Revisited



Git Server

- Authentifizierung und Autorisierung sind umgesetzt
  - Die Haupt-Branches der Flows (main/master) sind vor Entwickler-Pushes geschützt
  - “protected Branches”
- Ein Entwickler signalisiert die Fertigstellung eines Features durch Erstellung eines “Pull Requests”
  - Pull Request = Merge Request